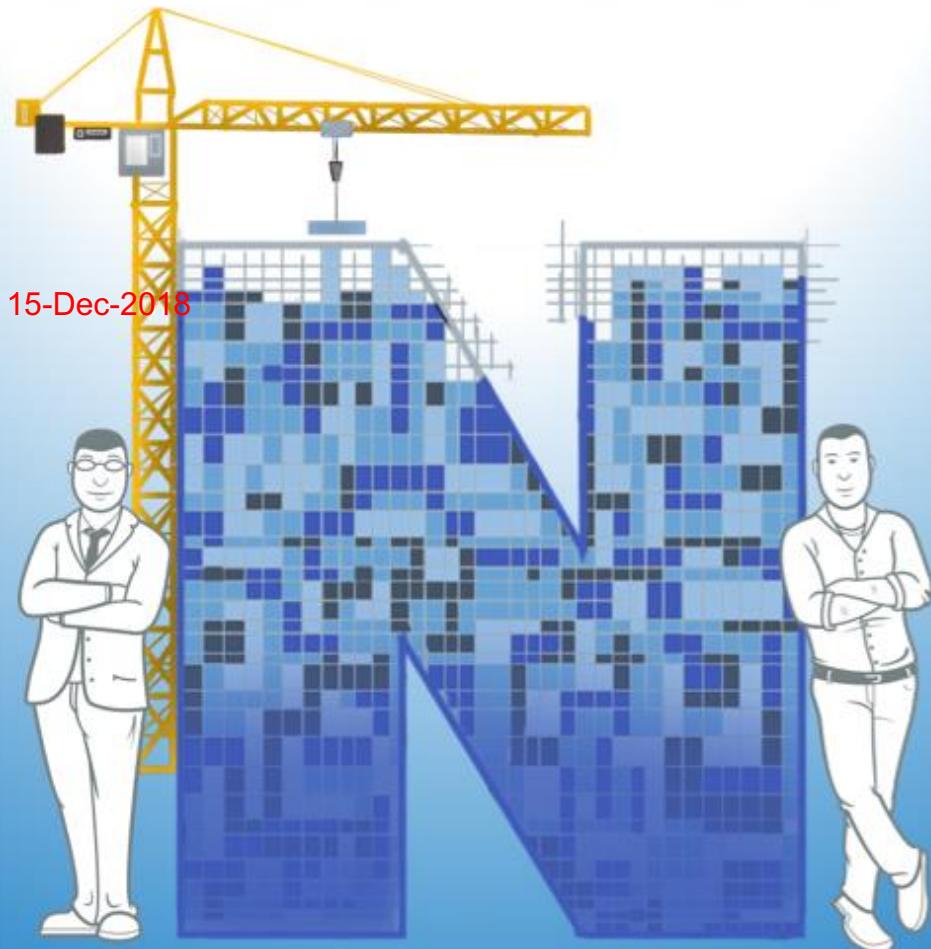


The NativeScript Book

building mobile apps with skills you already have



Mike Branstein | Nick Branstein

The NativeScript Book

building mobile apps with skills you already have

**MIKE BRANSTEIN
NICK BRANSTEIN**



©2017 by The Brosteins. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

NativeScript® is a registered trademark owned by Progress Software Corporation and is only used for reference. Progress Software Corporation does not sponsor or endorse this book.



Development editor: Toni Arritola
Technical editor: TJ VanToll
Technical reviewer: Alain Couniot
Cover designer: David Bjarnson
Reference drawings: David Bjarnson

brief contents

PART 1: THE BASICS

- 1 Why NativeScript*
- 2 Your first app*
- 3 Anatomy of a NativeScript app*

PART 2: STRUCTURING YOUR APP

- 4 Pages and navigation*
- 5 Understanding the basics of app layouts*
- 6 Using advanced layouts*
- 7 Styling NativeScript apps*

PART 3: REFINING YOUR APP

- 8 Working with data*
- 9 Native hardware*
- 10 Creating professional UIs with themes*
- 11 Refining the user experience*
- 12 Deploying an Android app*
- 13 Preparing an iOS app for distribution*
- 14 iOS security and building your app with Xcode*

PART 4: ANGULAR AND NATIVESCRIPT

- 15 Creating a NativeScript app with Angular*
- 16 Using Angular components and routing*

17 Angular data binding and services

APPENDIX

A Android emulator tips

B Creating custom UI controls

C NativeScript CLI Quick Reference

D NativeScript Conventions

contents

Part 1: The Basics 1

Why NativeScript 2

1.1	Introducing NativeScript	3
1.2	<i>What you'll learn in this book</i>	5
1.3	<i>What NativeScript means to mobile development</i>	5
1.4	<i>How NativeScript works</i>	8
1.5	<i>Summary</i>	9

Your first app 11

2.1	<i>Hello world</i>	12
2.2	<i>NativeScript apps</i>	19
2.3	<i>Establishing your development workflow</i>	22
2.4	<i>Quick reference</i>	25
2.5	<i>Summary</i>	26
2.6	<i>Exercise</i>	26
2.7	<i>Solutions</i>	26

Part 2: Structuring Your App 28

Anatomy of a NativeScript app 29

3.1	<i>Exploring the structure of a NativeScript app</i>	29
3.2	<i>Understanding app startup</i>	41
3.3	<i>Style guide and app organization</i>	43
3.4	<i>Summary</i>	47
3.5	<i>Exercise</i>	47
3.6	<i>Solutions</i>	47

Pages and navigation 48

4.1	<i>Creating a multi-page app</i>	49
4.2	<i>Creating another app page</i>	55
4.3	<i>Navigating between app pages</i>	58
4.4	<i>Summary</i>	65
4.5	<i>Exercise</i>	65
4.6	<i>Solutions</i>	65

Understanding the basics of app layouts 67

5.1	<i>Understanding NativeScript layouts</i>	67
5.2	<i>Stack Layout</i>	70
5.3	<i>Summary</i>	86
5.4	<i>Exercise</i>	86
5.5	<i>Solutions</i>	87

Using advanced layouts 88

6.1	<i>Introducing the grid layout</i>	88
6.2	<i>Adding content to a grid layout</i>	90
6.3	<i>Controlling grid layout rows and columns</i>	98
6.4	<i>Summary</i>	107
6.5	<i>Exercise</i>	107
6.6	<i>Solutions</i>	107

Styling NativeScript apps 108

7.1	<i>Using cascading style sheets</i>	109
7.2	<i>Adding images to an app</i>	122
7.3	<i>Summary</i>	138
7.4	<i>Exercises</i>	138
7.5	<i>Solutions</i>	138

Part 3: Refining Your App 140

Working with data 141

8.1	<i>Databinding</i>	143
8.2	<i>Observables in action</i>	145
8.3	<i>Observable arrays</i>	163
8.4	<i>Action bar</i>	168
8.5	<i>Summary</i>	172
8.6	<i>Exercise</i>	172
8.7	<i>Solutions</i>	172

Native hardware 174

9.1	<i>The file system module</i>	175
9.2	<i>Camera</i>	187
9.3	<i>Using GPS and location services</i>	199
9.4	<i>Summary</i>	204
9.5	<i>Exercise</i>	204
9.6	<i>Solutions</i>	204

Creating professional UIs with themes 206

10.1	<i>Themes</i>	206
10.2	<i>Using text classes, alignment, and padding</i>	209
10.3	<i>Styling buttons</i>	211
10.4	<i>Styling list views</i>	213
10.5	<i>Working with images</i>	215

10.6	<i>Styling data-entry forms</i>	216
10.7	<i>Summary</i>	222
10.8	<i>Exercises</i>	222
10.9	<i>Solutions</i>	222
Refining user experience		224
11.1	<i>Building professional UIs with modals</i>	224
11.2	<i>Adding tablet support to an app</i>	238
11.3	<i>Refining the tablet-specific user experience</i>	256
11.4	<i>Summary</i>	261
11.5	<i>Exercises</i>	262
11.6	<i>Solutions</i>	262
Deploying an Android app		264
12.1	<i>Customizing Android apps with the App_Resources folder</i>	264
12.2	<i>AndroidManifest.xml customizations</i>	267
12.3	<i>Launch Screens</i>	275
12.4	<i>Building your app</i>	286
12.5	<i>Summary</i>	289
12.6	<i>Exercise</i>	290
12.7	<i>Solutions</i>	290
Preparing an iOS app for distribution		291
13.1	<i>Transforming your app code into an iOS app</i>	292
13.2	<i>Finalizing your app</i>	294
13.3	<i>Summary</i>	308
13.4	<i>Exercises</i>	308
13.5	<i>Solutions</i>	308
iOS security and building your app with Xcode		309
14.1	<i>Building your app</i>	309
14.2	<i>Summary</i>	339
14.3	<i>Exercises</i>	340
14.4	<i>Solutions</i>	340
Part 4: Angular and NativeScript		341
Creating a NativeScript App with Angular		342
15.1	<i>Why Angular</i>	343
15.2	<i>Using NativeScript with Angular to recreate the Pet Scrapbook app</i>	347
15.3	<i>TypeScript</i>	350
15.4	<i>NativeScript Angular integration</i>	352
15.5	<i>Understanding NativeScript-with-Angular app startup</i>	352
15.6	<i>Summary</i>	357
15.7	<i>Exercise</i>	357

15.8 *Solutions* 357

Using Angular components and routing 358

16.1 *Creating static components* 358
16.2 *Navigating between components with routing* 368
16.3 *Summary* 378
16.4 *Exercise* 378
16.5 *Solutions* 378

Angular databinding and services 381

17.1 *Databinding with Angular* 381
17.2 *Creating and using services* 386
17.3 *Databinding events* 391
17.4 *Advanced databinding* 399
17.5 *Loading components as modal dialogs* 404
17.6 *Summary* 415
17.7 *Exercise* 415
17.8 *Solutions* 415

Appendix 417

Android emulator tips 418

A.1 *Emulator speed* 418
A.2 *Using Genymotion* 418

NativeScript CLI quick reference 419

B.1 *Creating apps* 419
B.2 *Adding the Android and iOS platforms* 420
B.3 *Building apps* 420
B.4 *Deploying apps* 421

NativeScript conventions 423

C.1 *Understanding NativeScript conventions* 423

Creating custom UI controls 435

D.1 *Introducing reusable, custom UI controls* 435
D.2 *Using custom UI controls* 453
D.3 *Summary* 460
D.4 *Challenge* 461

get the code

You can find the code in this book online on Github.

Visit <https://github.com/mikebranstein/TheNativeScriptBook> to get the code from each chapter.

preface

Finding the courage to take that first step and dive into something new like writing mobile apps can be intimidating and overwhelming. We understand, because that was us. We had so many questions and anxiety about where to start, and felt there was a huge wall to climb. Should we write native apps with Objective-C, Swift, or Java? If not, is Cordova or Xamarin the right choice? If we pick Cordova, will our apps be slow? How cross-platform is Xamarin really? Should I get a Mac so I can build iOS apps, or just stick with Android (for now)?

After a few years, our team had played the field: we decided to go cross-platform and had built Cordova and Xamarin apps. But, there was something missing. Our Cordova apps felt clunky and slow. Xamarin was expensive, and we found ourselves writing a lot of platform-specific code for Android and iOS. We could settle for Cordova or Xamarin, but really wanted a cross-platform approach that was inexpensive, performant, and allowed us to reuse some (or all) of our web development skills.

Then, in January 2015, we were introduced to NativeScript. It was early in its life, and the beta hadn't been opened to the public yet. We met with the NativeScript development team and Valentin (Valio) Stoychev walked us through NativeScript, start to finish. We were skeptical at first. Wouldn't you be? NativeScript promised a lot: native UI, near-native performance, full access to the underlying native APIs, and it was open source.

That summer, Nick, our co-worker Justin Tindle, and I hosted the Summer of NativeScript: a 3-session meetup group aimed at introducing others to NativeScript. About a dozen people showed up for the meetup each session, and we learned a lot about NativeScript together. We were surprised how easy it was to get started with NativeScript. The love affair had started.

Over the following months, I jumped headfirst into NativeScript. On week nights, I found myself writing apps for my boys, Charlie and Wesley. The apps were simple, but they let us enhance our creative play. One of my favorites was the Pokémon reference manual. Another favorite was the soundboard, where we could make their tablets say silly sayings like, "Watch out! There's a bigfoot behind you," and, "There's a yeti on your head." The soundboard is especially notable because the boys could invent a ridiculous phrase, and I could quickly add it to the app.

Later in the fall of 2015, our friend, Ed Charbeneau, introduced us to TJ VanToll, Jen Looper, and the rest of the Progress DevRel team. A NativeScript community was growing, and there was some interest in a book. After a few Sunday mornings at Starbucks, *The NativeScript Book* was born.

Our original concept was simple. *The NativeScript Book*: how to write professional mobile apps with JavaScript, CSS, and XML. 12 chapters and we'd be done. But then, something happened, and it happened fast. Angular 2 (which we now know as Angular). In October 2016, (when we were about half-way through the book), Angular 2 was released, and so did a new way to write NativeScript apps: NativeScript with Angular. This was huge. Bigger than huge. Gargantuan. The most popular JavaScript front-end framework could be used to write native mobile apps.

At first, we were reluctant to include Angular in our book, because the book was about NativeScript, not Angular. Even more important, we'd seen new NativeScript developers try to dive right into

NativeScript with Angular, and it was confusing and hard to understand which part of the app was NativeScript and which part was Angular.

Over time, we changed our minds. But, let's set the record straight: we didn't cave to the pressure. We found a way to teach the differences between [vanilla NativeScript](#) and [NativeScript with Angular](#). An updated version of The NativeScript Book was born, and an additional 5 chapters were added: three on Angular and two because I kept wanting to add more content (at least that's what Nick says).

That brings us to the present—17 chapters, covering vanilla NativeScript (without Angular) and NativeScript with Angular. We're proud and feel The NativeScript Book is a well-written book about a compelling technology. We also think you'll enjoy reading and learning how to build mobile apps with NativeScript. And, if your experience with NativeScript is like ours, you won't stop using it when you're finished with this book.

Thank you and enjoy!

acknowledgements

Before embarking on this journey we've called The NativeScript Book, we talked to a lot of authors. Every one of them said writing a book takes a lot of time and dedication. It's hard to place yourself in their shoes and visualize how much work constitutes a lot. Now we know. This book is the single most consistent and long-term project we've worked on together, and we owe it to each other to thank one another. But, there are so many more people we'd like to thank for helping us on this journey.

We'd like to acknowledge our development editor, Toni Arritola. Thank you for spending your Sunday afternoons with us. Thank you for talking us off the cliff when we disagreed, and thank you for talking Mike down when he insisted upon throwing Nick off the cliff. It's amazing how we could spend hours working on making a paragraph sound just right, but then you'd come along, switch the order of a few words, and make it sounds a million times better. We'd also like to thank our employer, KiZAN Technologies, for supporting us.

Another huge thank you goes to TJ VanToll, our technical editor. Even though your day job is NativeScript, you still found time nights and weekends for The NativeScript Book. You sent us down the right path when we couldn't decide which way to go. The support and encouragement you gave us kept us going. We also want to recognize all our friends at Progress Software: Ed Charbeneau (who first suggested we talk about a book), Jen Looper (for building an amazing NativeScript community), Dan Wilson (for giving us a voice and platform to evangelize NativeScript), and the rest of the DevRel team. Without the support of Progress, we couldn't have made this happen.

A special thank you is reserved for the NativeScript product team, who have built this open source, cross-platform framework – from scratch. NativeScript is something special. You know it, and the community knows it. Without your dedication, careful planning, and expertise, none of this would have been possible. We applaud and thank you.

We'd also like to thank the reviewers who took the time to read our book throughout its lifecycle. You provided us with invaluable feedback. Thank you to our co-workers, Justin Tindle, Jason Dailey, and Joshua Martin, who helped us understand how other developers see NativeScript, and steer us in the right direction. Special thanks to Alain Couniot, technical proofreader, for his careful review of the code one last time, shortly before we published.

Hi, it's Mike. I'd like to extend a thank you to my wife, Abigail, and two sons, Charlie and Wesley. Yours was the biggest sacrifice of all. You gave up a lot of late nights, early mornings, and weekend afternoons so I could do this. Your encouragement helped me succeed. I love you.

Nick here, I'd like to thank my wonderful wife Jen for your continuous support whether it be through cookies, coffee, or words of encouragement, all of which helped me succeed. I love you. I'd also like to thank my brother Mike for putting up with me throughout the entire book writing process and special thanks for pushing me to step out of my comfort zone and write a book.

about this book

~~NativeScript in Action was written to show you how it can be easy to get started with mobile app development using NativeScript, an open source framework for writing native mobile apps with Angular, TypeScript, or JavaScript. The first half of the book teaches you about the core of NativeScript apps: using JavaScript, XML, and CSS to build native, cross-platform apps without web views. In the second half, you'll learn how to build NativeScript apps with Angular and TypeScript.~~

~~It's a journey on learning how to write mobile apps with NativeScript. Finding the courage to take that first step and dive into something new like mobile apps can be intimidating and overwhelming. With NativeScript in Action at your side, you'll be able to walk away and build your first app, even if you're new to mobile app development.~~

Who should read this book

~~NativeScript in Action is for developers with familiarity with JavaScript, CSS, and XML. You don't need to be an expert in these technologies or even consider yourself an intermediate developer to become a mobile developer with NativeScript. So, if you've been developer for one to two years, you'll find that NativeScript will be easy to understand and jump right into.~~

How this book is organized

This book is presented in four parts, with 17 chapters.

In part 1, you'll learn the basics of NativeScript.

- Chapter 1 introduces you to NativeScript, describing what it is, why it's important, and why it's different from other mobile app development frameworks. It concludes with a brief overview of how a NativeScript app works.
- Chapter 2 goes deeper into how NativeScript apps run on mobile devices using a JavaScript virtual machine. You'll also learn how to create, compile, and run an app using the NativeScript command line interface (CLI).
- Chapter 3 wraps up part 1 and explores the structure of a NativeScript app. You'll learn about file and folder structure, various file-naming conventions, and how to organize your NativeScript apps.

Part 2 covers the essentials of creating and navigating between app pages. Throughout this part, we begin your introduction to various NativeScript UI elements used to organize pages and display text and images to users.

- Chapter 4 uses HTML applications as a point of reference to describe how NativeScript apps use the concepts of pages and navigation between pages. You'll also learn how to create pages and navigate between them.

- Chapter 5 describes the most widely used way to organize UI elements on a page.
- Chapter 6 builds on the previous chapter and introduces various other ways to organize UI elements.
- Chapter 7 closes out part 2 with an overview of styling NativeScript apps with CSS. You'll also learn how to integrate images into Android and iOS apps that target various screen resolutions and DPIs.

In part 3, you'll develop the Pet Scrapbook, a fully functional app that stores information and images of pets in a scrapbook.

- Chapter 8 introduces you to the Pet Scrapbook, the concept of dynamic data and data-driven UIs. You'll learn how to use data binding to build data-driven apps that can affect both text and UI elements.
- Chapter 9 teaches you how to use native mobile device hardware like the file system, camera, and GPS in NativeScript apps.
- Chapter 10 describes how to make NativeScript apps more professional and visually appealing by using themes. You'll learn how to leverage the NativeScript theme plugin to style the Pet Scrapbook UI to look consistent across Android and iOS devices.
- Chapter 11 continues the discussion of professional apps by introducing modal dialogs. You'll also learn how to quickly add tablet support to the Pet Scrapbook.
- Chapter 12 covers how to prepare NativeScript apps for the Google Play store. You'll learn about Android-specific settings, app icons, splash screens, and creating an Android app that is ready for store deployment.
- Chapter 13 mirrors chapter 12, but focuses on preparing NativeScript apps for the iTunes store. Preparing iOS apps is more involved than Android, so this chapter begins the discussion by teaching you how to create iOS-specific app icons, launch screens, and use the CLI to create an Xcode project.
- Chapter 14 finishes the discussion of preparing apps for the iTunes store, explaining iOS app security, and how to use Xcode to build and upload a NativeScript app to the store.

The end of part 3 marks the final version of the Pet Scrapbook, a complete NativeScript app, ready for store deployment and written using JavaScript, CSS, and XML. In part 4, we build the same NativeScript app, but using Angular and TypeScript.

- Chapter 15 introduces you to Angular and why you may want to create NativeScript apps with Angular. You'll build your first NativeScript-with-Angular app and learn how its structure differs from a traditional NativeScript app.
- Chapter 16 introduces Angular components, which represent pages in a NativeScript-with-Angular app. You'll learn how to create components and navigate between them using routing.
- Chapter 17 covers using Angular data-binding syntax to create dynamic UIs. You'll also learn how to use Angular service classes and navigate between components using modal dialogs.

Generally, we recommend reading the book from start to finish. Parts 1 through 3 help you learn and understand what NativeScript is and how it works. Part 4 builds on these concepts, and it's helpful to have

a solid understanding of NativeScript and Angular (separately) before jumping into NativeScript and Angular together.

About the code

This book contains many examples of source code, both in numbered listings and inline with normal text. In both cases, source code is formatted in a fixed-width font like `this` to separate it from ordinary text. Sometimes code is also in bold to highlight code that has changed from previous steps in the chapter, such as when a new feature adds to an existing line of code.

In many cases, the original source code has been reformatted; we've added line breaks and reworked indentation to accommodate the available page space in the book. In rare cases, even this was not enough, and listings include line-continuation markers (`→`). Additionally, comments in the source code have often been removed from the listings when the code is described in the text. Code annotations accompany many of the listings, highlighting important concepts.

Source code for the examples in this book is available for download from Mike's GitHub repository at <https://github.com/mikebranstein/TheNativeScriptBook>.

Other online resources

We wrote a lot of code for NativeScript in Action, and every listing, screenshot, icon, and app is online. You can follow along with each chapter in Mike's GitHub repository at <https://github.com/mikebranstein/TheNativeScriptBook>.

about the authors

Mike and Nick Branstein are brothers, and are collectively known as *the Brosteins* in the development community. They are .NET and JavaScript developers, consultants, and technology evangelists. They blog about technology, architecture and development for the cloud, development tools and techniques, application lifecycle management, gaming, the web, and team building at their blog <https://brosteins.com>.

dedication

Mike dedicates this book to his wife Abby and sons, Charlie and Wesley. Together, we've created some super cool apps with NativeScript. I love your creativity and goofiness. And, don't forget to watch out: there's a bigfoot standing behind you!

Nick dedicates this book to:

- His wife Jen for being the best wife there is
- His parents Sue Ann and Gary for helping mold me into the person I am today
- Pittens and Nibbles, may your journeys on the rainbow bridge be wondrous

Part 1: The Basics

1

Why NativeScript

This chapter covers

- What is NativeScript
- What NativeScript means to the mobile development world
- How NativeScript works

In the early days of mobile apps (pre-iPhone), not much emphasis was placed on methodologies for writing code once and deploying it to multiple platforms. Developers just wanted to get an app out to the Apple or Google Play stores as fast as possible. And if that meant their app didn't support both platforms, that was a reasonable sacrifice.

Fast forward to today: the mobile world is continually changing, making it more and more difficult to keep up with the latest devices. As developers create app, they need to reach the largest audience possible: focusing on a single platform just isn't an option anymore. Apps need to be available across platforms and devices. To keep up with ever-changing environment, developers are placing a premium on any technology that enables them to simplify the mobile app development process.

Today, developers have various choices for writing mobile apps that can target multiple platforms from a single code base. NativeScript is one of those choices, but it isn't the only one. You may have heard of others like PhoneGap, Xamarin, and React Native. Each of these frameworks is capable of writing code once and deploying it to both Android and iOS, but we're not here to debate the merits of one framework over another. Instead, we want you to learn how to write professional cross-platform mobile apps using skills you likely already have. If you're a beginner who knows the basics of writing web apps with HTML, JavaScript, and CSS, or seasoned expert, you can write a mobile app with NativeScript.

As you read this book, we'll show you how to write cross-platform apps from a single code base using the structured approach that NativeScript offers. When you're finished, you'll have the skills to create your own mobile apps for Android and iOS with your choice of technologies: HTML, JavaScript, and CSS or Angular, TypeScript, and CSS.

NOTE If you're not familiar with Angular or TypeScript, that's ok. The last 3 chapters of this book are dedicated to teaching you what they are and how they can be used to create mobile apps.

We've worked with a lot of developers learning NativeScript for the first time, and many of them want to jump straight into NativeScript with Angular. If that sounds just like you, go for it, but proceed with caution. If you're not familiar with Angular, learning both NativeScript and Angular at the same time can be confusing because the lines between what's NativeScript and what's Angular will be blurry. So, we recommend you learn about plain-old-vanilla NativeScript first. Follow along with our exercises in the first 3 parts of this book, then jump into NativeScript with Angular.

Before we get ahead of ourselves, let's back up and look at NativeScript in more detail.

1.1 Introducing NativeScript

NativeScript is an open-source framework for building cross-platform mobile apps for iOS and Android, created and maintained by **Telerik**. NativeScript differs from other mobile frameworks in many ways, the largest being that it is a cross-platform framework that can create native mobile apps with a single code base. Additionally, NativeScript offers a lot of features that make it easy to get started and leverage skills you may already have:

- Leverages your existing knowledge of HTML, JavaScript, and CSS (you don't have to know Objective C, Swift, or Java)
- All your code is written once
- Access to native platform APIs for Android and iOS
- An opinionated way to create apps that helps structure your code base
- Natively integrates with Angular (but doesn't have to)

Sometimes learning a new language is a barrier to entry into a new world. When creating NativeScript apps, you'll leverage your existing knowledge of HTML applications so you can quickly create an app targeting multiple platforms (Android and iOS). Because you already have these skills, you'll find that creating NativeScript apps can be quick. And even better, you won't have to learn Objective C, Swift, or Java.

1.1.1 How NativeScript apps are written

NativeScript apps are written in a combination of JavaScript, XML, and CSS, as shown in figure 1.1.

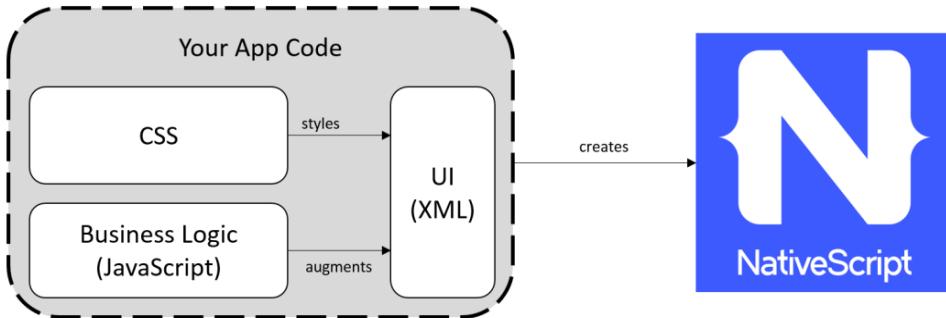


Figure 1.1 JavaScript, CSS, and XML combine to create a NativeScript app.

When you write NativeScript apps, your code has 3 parts: JavaScript, XML, and CSS. The JavaScript component runs business logic, accesses data, or controls the flow of the app. The XML portion defines the user interface (UI), and CSS is used to style the UI, much like an HTML application.

The structure and code of NativeScript apps closely resemble HTML applications, but this is where the similarities end. NativeScript is unique in the cross-platform mobile app space because it allows you to write your UI (XML) code once. When run, the UI code renders native UI elements in the app. For example, on iOS UI elements are rendered as native iOS buttons, dropdowns, lists, and so on. Likewise, on Android UI elements are rendered as native Android components.

Figure 2.1 shows the native rendering of an iOS button, written in NativeScript.

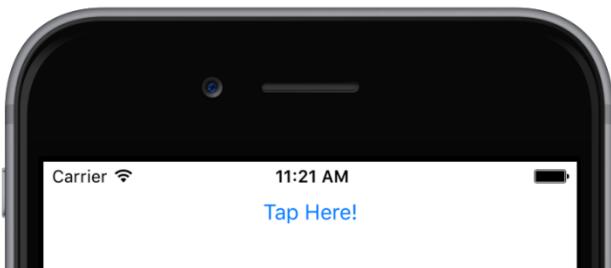


Figure 1.2 A native button in an iOS app written in NativeScript.

You'll notice that it looks just like an iOS button. And, that's because it *is* an iOS button. All NativeScript UI elements are native iOS and Android UI elements.

In other cross-platform frameworks, you may have to spend time writing specific view code for specific platforms. But, the ability to write your UI code once and have it render as native UI components is a feature that sets NativeScript apart from other frameworks.

Another unique feature of NativeScript is that you have access to Native APIs.

NOTE Yes, several of the previously mentioned frameworks also let you access Native APIs. But, as you'll learn throughout this book, the way NativeScript runs and accesses the Native APIs of Android and iOS is much different. Even though you write NativeScript code in JavaScript, you have access to every native API function, feature, and hardware component device your app runs on.

As you continue to learn about NativeScript, you'll see how NativeScript executes all your code as native code running on the device. This allows you to take advantage of the performance gains of writing native code without having to learn or write Objective C, Swift, or Java!

Now that you have a high-level familiarity with NativeScript, let's look at what you'll learn in this book.

1.2 What you'll learn in this book

At this point, you're beginning to get an idea of the technologies you'll use to write NativeScript apps (HTML, JavaScript, and CSS). Not a whole lot, right? If you already have these skills, you may be wondering why you should keep reading. In this book, we'll teach you how to take these skills and apply them to create professional-looking mobile apps.

What do we mean by professional? Professional can mean different things to different people. You might think just showing up to work on time is professional, while your friend may think professional means wearing a suit to an interview.

NOTE To us, professional means creating a single, maintainable code base for your app so it can continue to grow over time.

Creating a professional app is also about using NativeScript's features so your app looks and feels native on the platform that it is running on.

While you're learning to create professional apps, you'll discover how NativeScript apps are structured and how to access native hardware components such as the camera, GPS, and location services.

But, before we jump straight into code, it's important that we put NativeScript into perspective, so you understand how it works.

1.3 What NativeScript means to mobile development

Think back 15 years (if you can), when you were carrying around a Windows 6 mobile phone or geeking out over the latest Samsung Blackjack: this was before Android and iOS. There were just fewer platforms and devices back then. Today, new devices come out monthly. And because of this increasing rate and variety, the development community has begun to look for more efficient ways to write mobile apps that target all the platforms.

1.3.1 Different types of mobile apps

Mobile apps fall into one of four major categories: native, hybrid, cross compiled, and just-in-time (JIT) compiled (table 1.1).

Table 1.1 Different mobile app types and their popular frameworks

Mobile App Type	Framework
native	Android, iOS
hybrid	PhoneGap/Cordova
cross compiled	Xamarin
JIT compiled	NativeScript

DEFINITION Just-in-time (JIT) compiled apps are apps that are compiled at runtime versus being compiled before the execution of the app. For example, in a just-in-time app, your source code is not compiled to native machine code until the absolute last minute, or immediately prior to executing each statement.

Excluding native apps, the other three app types in table 1.1 have the same goal: write your app code once and deploy it to multiple platforms (which is what people mean when they say cross-platform).

Even though the cross-platform frameworks listed above achieve similar results, they do so in a variety of ways. Figure 1.3 shows the differences between the different types of mobile apps and how they run on devices.

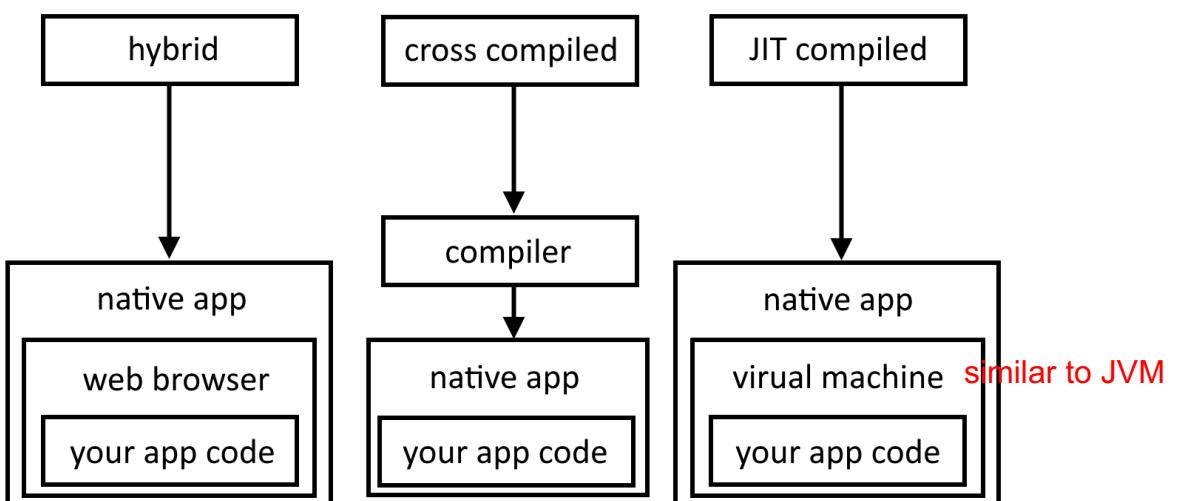


Figure 1.3 How types of mobile apps run on a device

In figure 1.3, you can see how each type of mobile app uses a different mechanism to run on a mobile device. Hybrid mobile apps are essentially webpages run inside of a web browser. Cross compiled apps are compiled into, which transforms them into a native app. Lastly, JIT compiled apps (like NativeScript),

run inside of virtual machine. For NativeScript apps, your app code runs inside of a *JavaScript virtual machine*.

DEFINITION A *JavaScript virtual machine* is a piece of software that runs JavaScript code.

If you're a .NET or Java developer, you're already familiar with running code in a virtual machine because both .NET and Java run code in a virtualized manner. The way NativeScript works is similar.

1.3.2 Why NativeScript is important

Besides JIT compilation, NativeScript has a variety of other differences when compared with other mobile app frameworks. We think the most significant difference is your ability to write truly native apps from a single code base and deploy it to both Android and iOS with no changes.

We've worked with other mobile app frameworks in the past, and in our opinion, NativeScript stands apart. In other frameworks, we've had to write a lot of *shim* code. This *shim* code acts like a piece of wood that's used to level a stove in your kitchen or to help frame a doorway. To continue the analogy, imagine you're installing a new door and door frame. Most doors are built to a standard width, height, and depth, and they fit almost right. But in all cases, you add a little shim here and a little shim there to get it to fit just right. This is what it's like when writing code in other frameworks: you add a bit of UI code to make a button display just right in the Android version of the app, and a little more UI code to make it look just right on iOS.

NOTE Hold on, we're not trying to paint the picture that NativeScript is perfect, because nothing is. But, NativeScript is compelling, and in our opinion, requires the least number of shims. In fact, the shims are so limited, you may never come across them when you're writing a line-of-business app. And when you do run across them, there's an extensive community of NativeScript experts ready and willing to help at <https://nativescript.org>.

GETTING TO MARKET FAST

So, what does this all mean: less shim code, write-once, deploy everywhere, and so on. Whether you're a business, an independent developer, or a casual enthusiast, you don't want to waste your time. And, these things (less shim code, write-one, and deploy everywhere) means you'll spend less time developing your app, giving you more time to innovate and release more features in less time.

1.3.3 What types of apps can be built with NativeScript

Now that you know a bit more about how NativeScript works, we think it's important that you know the type of mobile apps you can write with it. You'll recall that NativeScript apps run directly on the device and are interpreted by a *JavaScript virtual machine* running inside of the app. This means NativeScript apps aren't restricted from accessing native device APIs or hardware, so any app can be written as a NativeScript app.

WARNING Hold up. Just because you can doesn't mean you *should*.

Let's start by looking at app types that you *shouldn't* create with NativeScript.

GRAPHIC-INTENSIVE GAMES

Let's start off being clear: don't write graphic-intensive games with NativeScript.

Imagine you're developing the next big mobile game: Floppy Bunny, and Floppy Bunny requires a lot of graphical and computational power to render its intense 3D graphics. While NativeScript is very performant out of the box, there are likely better platforms made for the express purpose of creating highly-performant 3D games.

After all, NativeScript apps run inside of a JavaScript virtual machine, so there's an extra, albeit small, layer of abstraction between your app and the bare metal. To extract every bit of performance out of a device and make Floppy Bunny an overwhelming success, you should consider writing a native Android or iOS app.

LINE-OF-BUSINESS AND CONSUMER APPS

If you're feeling down because we shattered your hope of writing Floppy Bunny, don't worry. There are other types of apps that NativeScript is great for!

Unlike our game example, NativeScript is a perfect choice if you're developing a line-of-business app such as a news feed, companion app for a website, social media app, or even an app to control all the smart devices in your home! In fact, there's a wide variety of apps already written in NativeScript across dozens of industries. Check out a showcase of these apps at <https://www.nativescript.org/showcases>.

1.4 How NativeScript works

Writing native mobile apps using JavaScript, XML, and CSS isn't something you commonly hear about. Instead, you hear about writing native mobile apps in Objective C, Swift, or Java. NativeScript makes it possible to write native mobile apps with several components: the NativeScript runtime, core modules, JavaScript virtual machines, your app code, and the NativeScript command line interface (CLI). Figure 1.5 shows how these components work together to create native Android and iOS projects, which get built into native apps that run on a mobile device.

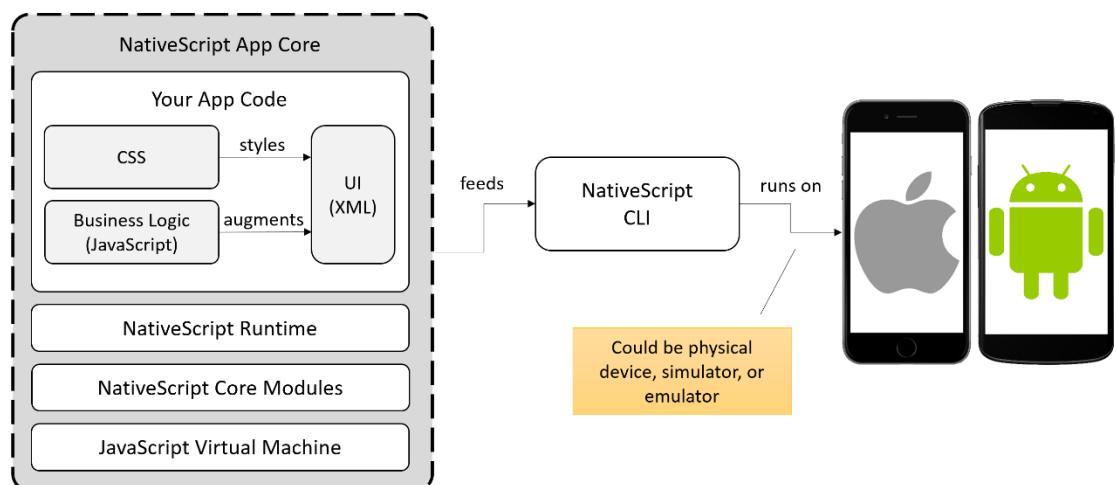


Figure 1.5 How NativeScript components and your app code work together to build and run native iOS and Android apps

We know there are a lot of boxes and lines in figure 1.5, and visualizing how these components work together at this point may seem overwhelming. Don't worry. We will go through each of the items later in this book. For now, let's get you started by explaining how everything works together at a high level.

Let's start with something you've already learned: your app code is written in JavaScript, CSS, and XML. After you've written your code, it interacts with the NativeScript runtime and the NativeScript code modules (API modules that you'll learn about in this book).

Finally, a tool known as the NativeScript CLI, bundles your code, the NativeScript runtime, and NativeScript core modules into a native app that contains a JavaScript virtual machine. This native app then runs on Android and iOS.

DIVING DEEPER

That's it! You just learned how NativeScript apps work at the 10,000-foot level, but let's dive a little deeper.

After creating your user interface (UI) using XML, you use CSS to style the UI (like CSS is used to style HTML apps). Then, you write JavaScript to augment your UI. Your JavaScript code will contain writing business logic that responds to events (like the app startup event) and interactions (like a button tap or finger swipe). These three pieces (UI written with XML, CSS, and business logic written with JavaScript) combine to create your app code.

By itself, your app code doesn't have everything it needs to run on a mobile device; you also need the help of three additional components: the NativeScript runtime, core modules, and a JavaScript virtual machine. We'll explore these components in future chapters, but for now, just remember that your app code and these three components form the core of your NativeScript app.

After you've developed your app code, it is fed into the NativeScript command line interface (CLI). The CLI is responsible for creating native Android and iOS projects and merging the NativeScript app core into each project. When run, the CLI invokes the native Android or iOS software development kits (SDKs) to build and compile a native app. The compiled app is then deployed (by the CLI) and runs on a physical device, simulator, or emulator.

As you can see, NativeScript's beauty lies in its universal nature: you don't have to spend time learning native programming languages like Objective-C, Swift, and Java because you can use JavaScript. Furthermore, the platform agnostic commands provided by the NativeScript CLI ensure you don't have to learn how the native tools and SDKs for Android and iOS work.

1.5 Summary

In this chapter, you learned that:

- NativeScript apps are written in JavaScript, XML, and CSS and run in a JavaScript virtual machine.
- Your app code works with the NativeScript runtime, core modules, and a JavaScript virtual machine to create the core of a NativeScript app.
- The NativeScript CLI abstracts away the complexities of native tools and SDKs, providing you with

a single platform-agnostic set of commands to build and deploy your app.

NOTE Before you continue, you'll need to get your development environment set up. Please refer to the official NativeScript installation instructions at <http://docs.nativescript.org/start/quick-setup>.

TIP If you are having difficulties getting the Android emulator setup and running, please see the Android Emulator Tips in appendix A.

3-Jan-2019

3 - Jan -2019

2

Your first app

This chapter covers

- The NativeScript runtime
- The NativeScript development workflow
- The NativeScript CLI
- Your first NativeScript app

In chapter 1, you were introduced to NativeScript. You learned that NativeScript provides you a way to write your app code once and deploy your app to multiple platforms (iOS and Android). You also learned that you can use your existing development knowledge of XML, JavaScript, and CSS to create NativeScript apps. Now it is time for you to take a closer look at NativeScript and write your first app!

TIP When learning to develop cross-platform mobile apps, you should choose a single device to test on during your initial development. This is important because you don't want to lose focus on the creation of your app by testing on too many platforms at once. As you finish developing each feature of your app, stop and test your application on various devices. Once you're satisfied the feature works across all platforms, then return to a single device for further development.

Throughout the book, we've chosen to develop and test our apps on an iPhone 6, so you'll see a lot of iPhone screenshots. When it makes sense, we'll include a side-by-side comparison of the same app code running on Android. Just because we're starting with an iPhone doesn't mean you have to as well: use the platform you're familiar with because it will make testing easier.

Before you get started developing your first app, let's take a closer look at the anatomy of a NativeScript app.

2.1 Hello world

It may be a bit cliché, but the hello world app is still a great way to get started in any new language (why should NativeScript be any different?). After creating your hello-world app in NativeScript, you will have gained the necessary knowledge to continue creating more robust apps like the store front app that you will be building in chapter 4. Let's start creating the hello-world app by using the NativeScript command line interface (CLI) tools.

2.1.1 NativeScript CLI

As you learned in chapter 1, the NativeScript CLI is a collection of tools you'll use to build and run NativeScript apps. The CLI is nothing more than an npm package that was installed into Node.js when you configured your development environment for NativeScript.

DEFINITION Node.js is a JavaScript virtual machine (VM) interface written to interact with desktop and server operating systems like Linux and Microsoft Windows. It was created in 2009, and allows developers to use JavaScript and write software that can be run cross-platform. Like web browsers and NativeScript, Node.js provides interface code to inform the JavaScript VM how interact with different operating systems. You can learn more about Node.js at <https://nodejs.org>.

The NativeScript CLI will be used throughout your development lifecycle, as shown in figure 2.1.

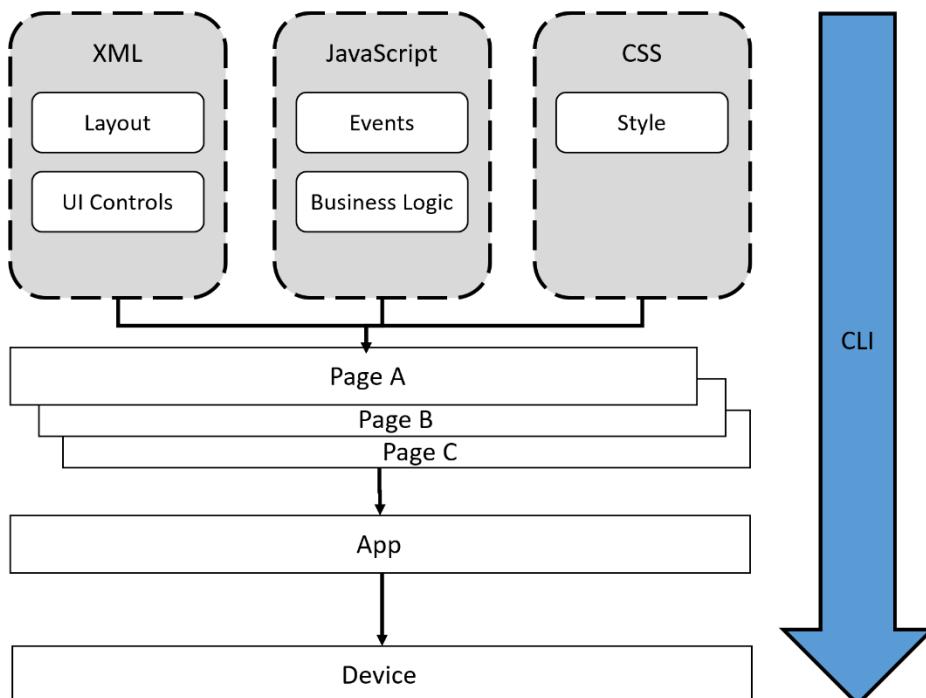


Figure 2.1 The CLI is used throughout the development lifecycle, transforming the XML, JavaScript, and CSS code you write into apps that can be deployed to a device.

NativeScript apps are a collection of code files written in XML, JavaScript, and CSS. Collectively, the UI, business logic, and styling form the pages of your app, like HTML application pages. Once you've written the pages for your app, you use the NativeScript CLI to transform your code into a native app. You then use the CLI to install and run the app in an emulator, simulator, or on a physical device.

You use the NativeScript CLI by invoking the `tns <sub-command>` command from your favorite terminal or command line window.

DEFINITION TNS stands for **Telerik NativeScript**. Telerik is the company behind the NativeScript project. The abbreviation TNS was chosen over NS to avoid confusion. The abbreviation NS is used throughout Mac OS X and iOS APIs, standing for NextStep.

Various sub-commands exist in the CLI, all aimed at making it easy to build, deploy, and run your app on Android and iOS platforms. We're not going to delve into the inner-workings of the CLI, but we feel it's important you understand how integral the CLI is to NativeScript development. Without the CLI, all you have is XML, JavaScript, and CSS files sitting around with no place to go. Throughout the book, you'll see us use the CLI for the following:

- Creating your app
- Adding a mobile platform to your application (Android or iOS)
- Building your code into a native application (.apk file for Android and .ipa application file for iOS)
- Deploying your application to a mobile device
- Using an emulator to test your code
- Running your application on a device connected to your computer

2.1.2 Using the CLI to scaffold your project

A core function of the CLI is to accelerate your development process by automating tedious tasks. One such task is creating a new NativeScript app. NativeScript apps have a very specific file and folder structure that needs to be in place, and the CLI makes this lengthy task easy through a process called scaffolding.

DEFINITION Scaffolding is the process of using a predefined template to generate files, folders, and code. This process is called scaffolding because it resembles construction scaffolding, which is a temporary structure on the outside of buildings, used by workers while they build the building. Just like construction scaffolding, code scaffolding creates an application's base structure that is then built-upon and modified to create a working app. Scaffolding tools are becoming more-prevalent in software development because they help reduce the time developers spend writing common, repeatable code constructs.

You may be familiar with similar scaffolding tools such as the project templates built into Visual Studio or another Node.js tool named Yeoman. Just like these tools, the NativeScript CLI's scaffolding process uses a collection of templates stored online as npm packages.

NOTE NativeScript templates are just npm packages. To find them, go to the npm website <https://www.npmjs.com>, and search for packages beginning with the prefix tns-template.

What is Yeoman?

Yeoman is a scaffolding tool for building modern web apps with Node. Just like templates in NativeScript, Yeoman templates are just NPM packages created and supported by the development community.

For more information about Yeoman you can visit <http://yeoman.io>.

You will be using the *hello world template* to create your first NativeScript app. The hello world template is NativeScript's default template used to create new apps. To get started with this template, load your favorite terminal. If you're using Windows you can use the command prompt or PowerShell; on Windows, I generally stick to using the command prompt. On Mac OS or Linux, you can load your favorite terminal program. When working on NativeScript apps on Mac OS, I use the default terminal.

During the installation of NativeScript, the `tns` command was added to your path or shell profile (depending on the OS you are using). Let's use it to create your first app by running `tns create HelloWorld`.

When you run the `tns create {app-name}` command, the NativeScript CLI creates a new folder (in the current folder) with the app name that you specified. After creating the folder, it scaffolds all the code needed to build and run a new NativeScript app! Figure 2.2 shows the resulting directory structure that is created for you after running the `create` command.

HelloWorld

- app
- node_modules
- platforms
- package.json

Figure 2.2 The file structure of your Hello World app.

Later in this book, we'll go into more detail and you'll learn about the folder structure of NativeScript apps. For now, note that all the code that you will be writing for your app will be in the `app` folder.

TIP You ran the `tns create` command with only one parameter: `HelloWorld`. Alternatively, you could have specified which template you wanted to use when scaffolding your project by using the `--template {template name}` argument. Without specifying the template name, the `create` command will use the default hello-world template. You could have achieved the same results by running: `tns create HelloWorld --template tns-template-hello-world`. In chapter 3, you'll learn more about templates and how the NativeScript CLI scaffolds apps.

2.1.3 Initial platform and development tools

Before running your hello world app, you'll need to decide whether you want to initially target iOS or Android. It's important to select one platform to target first, so you can focus on testing functionality in a single app and on a single emulator or simulator. Once you're confident your app works well on your first platform, you can test it on the second.

Deciding on your development platform may seem like a tough decision, but it is probably easier than you think because part of this decision may already be made for you. Table 2.1 shows the potential platforms you will be able to target (depending on your OS). Unfortunately, you can't target iOS unless you have a Mac, so if you're using Windows or Linux, Android is your starting platform. If you do have a Mac, the choice is yours. But, if you're still having trouble deciding, we recommend you start with a platform you're familiar with. If you have an iPhone, target iOS, otherwise target Android.

Table 2.1 Targetable platforms for each development OS

Development Machine OS	Targetable Platforms
Windows	Android
Mac OS	iOS, Android
Linux	Android

Don't worry if you are planning to use a Windows machine as your primary development machine and would like to build for iOS later. It is easy to target and build your app for different platforms in NativeScript. Targeting one platform from the start is the preferred workflow when working with NativeScript. When the time comes to get your app on another platform, you can always borrow a Mac from a friend (or better yet, buy an older refurbished Mac mini)!

In addition to choosing your target app platform, you'll need to decide on the development editor you are going to use. Whether you choose a basic text editor, Visual Studio, or another integrated development environment (IDE), there is no wrong answer. Choose your favorite development tool. One of our favorite editors that works well on both Windows, Mac OS, and Linux is Visual Studio Code.

TIP Visual Studio Code is completely free and can be downloaded from <https://code.visualstudio.com>. Visual Studio Code works well on both Mac, PC, and Linux. It is what we use to write NativeScript apps and is highly recommended for you while learning NativeScript. There is an official NativeScript extension for Visual Studio code that will assist you while creating your app with better intellisense,

debugging, and emulator support. The extension is available at <https://www.nativescript.org/nativescript-for-visual-studio-code>.

2.1.4 Adding and removing platforms

Now that you have started to create your first official NativeScript app, you are almost ready to get to testing! Before you start testing your app, you will need to make sure your hello world app is targeting the platform of your choice (or both). Adding and removing a platform is easy. If you need to add a platform, run the following commands. Figure 2.3 shows the resulting platforms folders:

```
tns platform add android  
tns platform add ios
```

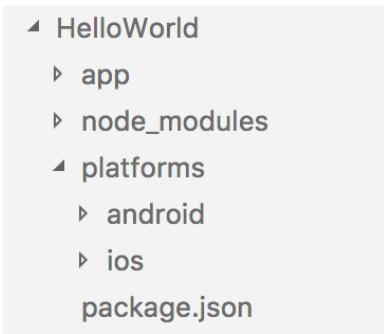


Figure 2.3 The resulting platforms folder on Windows, after you create your NativeScript app.

When you run the `tns platform add` command, the NativeScript CLI creates native Android and iOS projects and adds important NativeScript runtime libraries to each project. These libraries (along with the code you write) will eventually be bundled together into a native app and stored in a subfolder of the platforms folder.

Because new features are continually being added to NativeScript, the NativeScript runtimes are updated regularly. The NativeScript runtimes are included in the NativeScript npm package that you previously installed. We recommend that you keep your NativeScript version up-to-date while learning NativeScript with this book. You don't have to update the NativeScript runtime when a new version is available, but you may want to. You can check the version of NativeScript you have installed by running the `tns` command with the `--version` parameter.

```
tns --version
```

You can check the latest version of NativeScript on npm at <https://www.npmjs.com/package/nativescript> or using the npm command `npm view nativescript version`. Since NativeScript is just an npm package, you can update it using the npm command `npm install -g nativescript`.

NOTE Remember that NativeScript is an open-source project. You can read the version history and latest changes to NativeScript on Github at <https://github.com/NativeScript/NativeScript/releases>. If you're so inclined, you can contribute to the project by submitting a pull request.

After **updating** your NativeScript runtime you will want to update the runtime files in your app (but first you will need to remove the old files). The following commands show you how to remove the Android and iOS platforms from your project using the `tns platform remove` command.

```
tns platform remove android  
tns platform remove ios
```

The NativeScript CLI will tell you if removing the platform was successful or not. If the platform did not exist as a target in your project, the NativeScript CLI would not be able to remove it. Figure 2.4 shows the resulting empty platforms folder after running both commands.

```
mikeb-macbook-pro:platforms mike$ ls -al  
total 0  
drwxr-xr-x 2 mike staff 68 May 14 21:34 .  
drwxr-xr-x 6 mike staff 204 May 14 21:33 ..  
mikeb-macbook-pro:platforms mike$ █
```

Figure 2.4 Empty platforms folder after running the command to remove both platforms.

To update your app with the latest version of the NativeScript runtime for the Android or iOS platforms, run the following commands:

```
tns platform add android  
tns platform add ios
```

NOTE If you try to add iOS as a target platform when developing on a Windows or Linux machine, the NativeScript CLI will detect this and throw an error message. (Remember that Mac OS is a prerequisite for targeting iOS as a platform).

As you've probably figured out by now, NativeScript organizes platform-specific files underneath the platforms folder. When the platform add command is run, the latest runtime files will be copied into the corresponding folder underneath the platforms folder. For example, if you browse the android subfolder underneath platforms, you will notice quite a few files. The files that are copied consist of the following:

- Platform-specific files
- Platform configuration files

You will learn more about platform-specific files in a later chapter.

2.1.5 Running your app in an emulator

Now that you've learned how to target a platform, it's time to run your app in an emulator and test it! Use the following commands to launch your app, using either the Android emulator or the iOS simulator:

```
tns run ios --emulator  
tns run android --emulator
```

NOTE Make sure you are running the `run` command from inside the root of the `HelloWorld` project folder or else you will get an error.

When you use the `run` command, the **NativeScript CLI automatically builds a native Android or iOS app for you**. Figure 2.5 shows how the files in the platform-specific folders are combined with the code for your app to create a resulting native app that will run on Android or iOS.

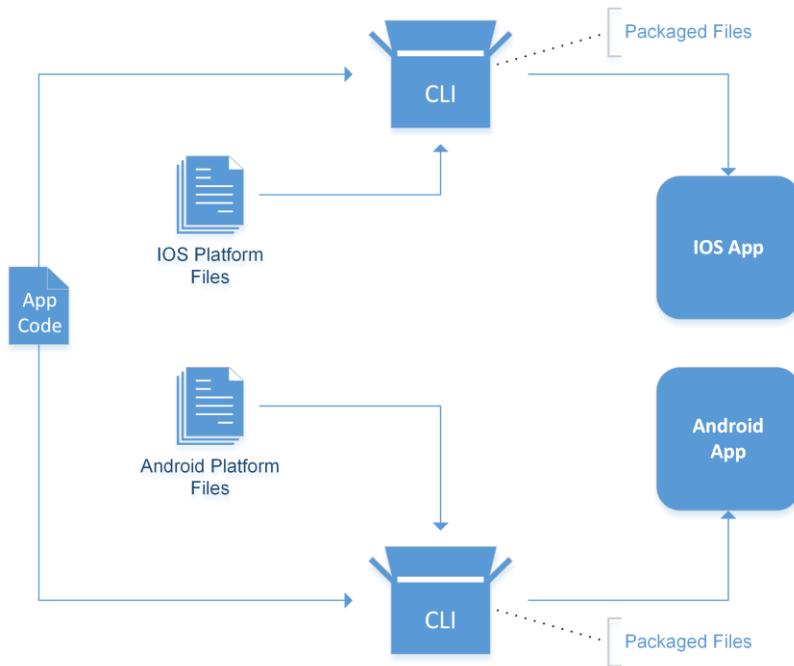


Figure 2.5 The CLI is used to package app code and platform-specific files into native Android and iOS apps

If you ran your app with the `iOS` parameter, the iOS platform files in the `iOS` folder were used to create the native iOS app. Likewise if you ran your app for `Android`, the NativeScript runtime for `Android` (in the `android` folder) was used. Figure 2.6 shows the resulting app running on a simulated iOS device.

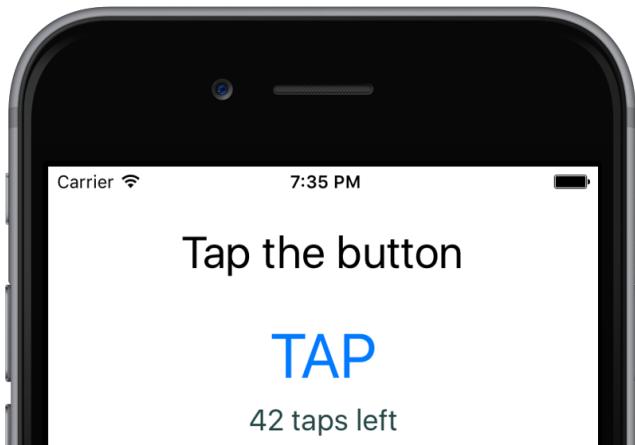


Figure 2.6 The hello world app running in the iOS simulator.

The resulting `hello-world` app that you created consists of a button that you can tap. Go ahead and click it and see what happens!

2.2 NativeScript apps

Writing a native mobile app using JavaScript, XML, and CSS isn't something that you commonly hear a developer talking about. Instead, you hear about writing mobile apps in Objective C, Swift, or Java. NativeScript makes it possible to write apps in JavaScript with several components: your app code, the NativeScript runtime and core modules, a JavaScript virtual machine (VM), and the NativeScript command line interface (CLI). Figure 2.7 shows how all the pieces of a NativeScript app fit together.

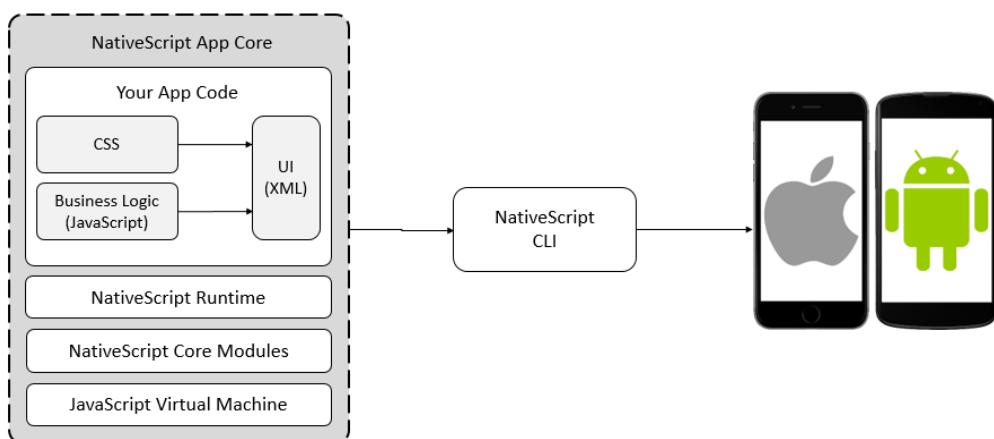


Figure 2.7 The NativeScript CLI takes 4 components (your app code, the NativeScript runtime and core modules, and a JavaScript VM) and bundles them together to create a native app.

The core of a NativeScript app comes from the code that you write (in XML, CSS, and JavaScript). The code that you write will also take advantage of the NativeScript core modules and NativeScript runtime. The NativeScript runtime and core modules are the libraries you will use when creating NativeScript apps. Your app code and these libraries run inside a JavaScript VM (we'll be going into more detail on this in a bit). Together these 4 components are bundled together by the NativeScript CLI to create native apps for Android and iOS.

2.2.1 The NativeScript runtime

When mobile apps are written in native code, they are compiled into a special application file and distributed to a mobile device. Figure 2.8 shows how native code is written and deployed to devices.

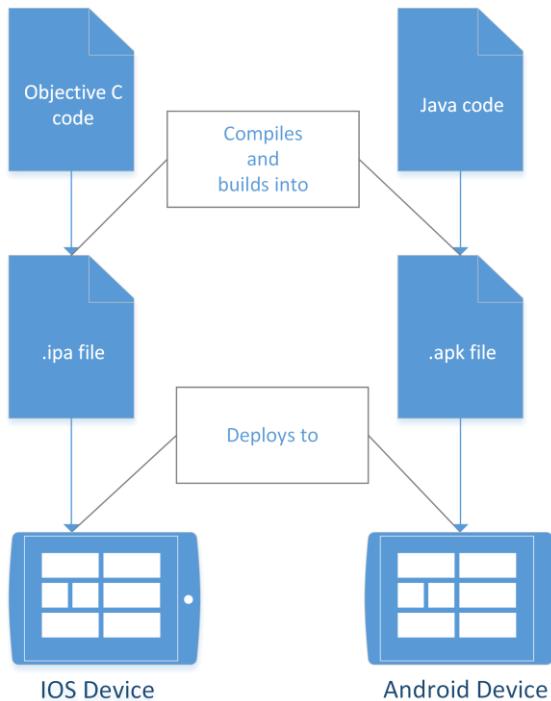


Figure 2.8 Native apps are written in Java (Android) and Objective-C (iOS) and compiled into .apk and .ipa files. These files are then deployed to a mobile device where they run.

For iOS, you write code in Objective C, which is compiled and built into an .ipa file that is distributed to a device running iOS. For Android, you write Java code, compile and build the code into an .apk file,

and distribute the .apk file to an Android device. When iOS and Android run their respective applications, devices natively run the compiled Objective C and Java code.

When you write an application in NativeScript, you write the application in JavaScript, which is not compiled, but is bundled into an application file along with the NativeScript runtime. Figure 2.9 shows how the app code is run on an Android or iOS device.

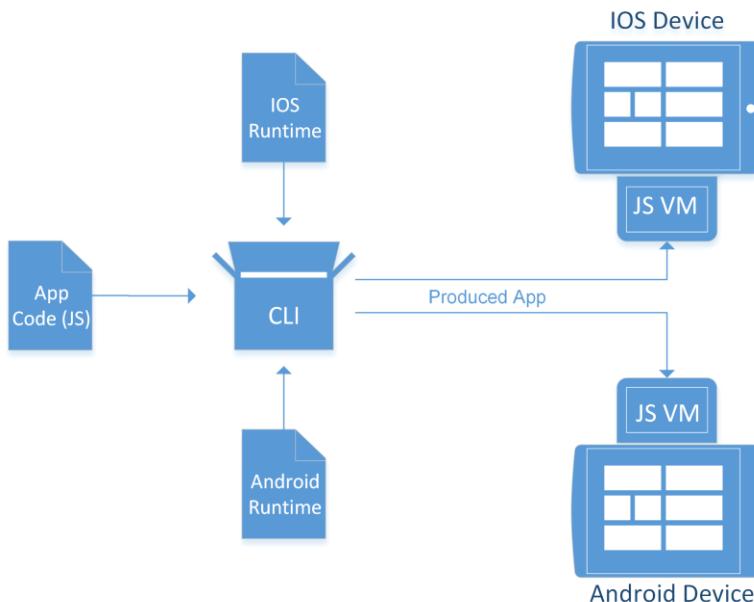


Figure 2.9 The CLI bundles your app code (written in JavaScript) and the iOS and Android runtimes to produce an app that runs inside of a JavaScript virtual machine on a device

As you can see, you write your app code one time using JavaScript and the NativeScript runtime ensures that it runs on Android or iOS. The NativeScript runtime is what issues the native calls on the device when your app is running. The runtime does this through a JavaScript VM that is bundled into your native app.

2.2.2 JavaScript virtual machines

JavaScript VMs are specialized pieces of software that take JavaScript code and run the code on an underlying environment. If you're familiar with web programming, you will have worked with a JavaScript VM in your web browser. Each browser runs a JavaScript VM, which in turn, runs JavaScript code you write. In simple terms, JavaScript VMs know how to do one thing: read and execute JavaScript code. For example, a VM knows how to read and execute JavaScript variables, FOR loops, and functions.

Because JavaScript VMs know only about JavaScript, they don't know about web browsers. Even though they're part of a browser, without help, they don't natively understand browser concepts such as

the *DOM* (*document object model*), the *document* object, or the *window* object. As a result, when a JavaScript VM is included inside of a web browser, the browser manufacturer creates interfacing code to teach the VM how to interact with a browser. This teaches the VM that the *document* and *window* objects exist, and represent the HTML document and the browser window, respectively.

2.2.3 JavaScript virtual machines in NativeScript

Now that you've been introduced to JavaScript VM, and you understand how a VM is bundled with browsers, you'll learn how NativeScript leverages a JavaScript VM. Similar to browsers, NativeScript apps are bundled with a JavaScript VM to read and run your JavaScript code. Just like a JavaScript VM in a browser, the JavaScript VM bundled with NativeScript apps know how to do one thing—read and execute JavaScript code.

But wait! If the NativeScript JavaScript VM only knows how to read and execute JavaScript variables, FOR loops, and functions, how does it know how to interact with a mobile device like an Android phone or iOS tablet? Without help, it doesn't, which is why the NativeScript development team has written **interfacing code** (called the *NativeScript core modules* and *NativeScript runtime*) to teach the JavaScript virtual machine about mobile device APIs such as Android and iOS.

DEFINITION The NativeScript **core modules** are a collection of libraries that you will learn about throughout this book. The libraries are what you will use to build your app and instruct the NativeScript runtime what you want your app to do on a device. The **core modules** consist of different libraries such as UI components (buttons, list views, labels), navigation, and the application.

DEFINITION The NativeScript **runtime** is interface code that bridges the gap between JavaScript code and the **native APIs** for Android and iOS. Just like browser manufacturers teach their JavaScript VMs about the *document* and *window* objects with interface code, the NativeScript runtime teaches its JavaScript VM about the underlying native device APIs.

2.3 Establishing your development workflow

Now that you have a deeper understanding of how NativeScript works, it is time to establish a development workflow for yourself. It is important to establish your development workflow before you start creating your app. Because you write your code only once and the NativeScript runtime runs your code as native code, you won't have to spend a lot of time worrying about how your app will run across the different platforms.

NOTE There are some features in Android that may not be in iOS and vice-versa. Because of this, you still may have to write some code that is specifically targeted at Android or iOS. Luckily, NativeScript provides several mechanisms for targeting code at specific platforms. You will learn more about platform specific code in chapter 3.

Not worrying about platform-specific implementations is a relief because it allows you to focus on features and functionalities of your app to make it better. And truthfully, I don't want to know the details of how buttons and text are displayed on Android versus iOS. That is what NativeScript is for!

Now, let's learn about the typical NativeScript development workflow. This basic process is something you'll do over and over while you're writing NativeScript apps. Figure 2.10 shows the three-step process of writing your app code, invoking NativeScript CLI commands to bundle your code into a native app, then testing your app.

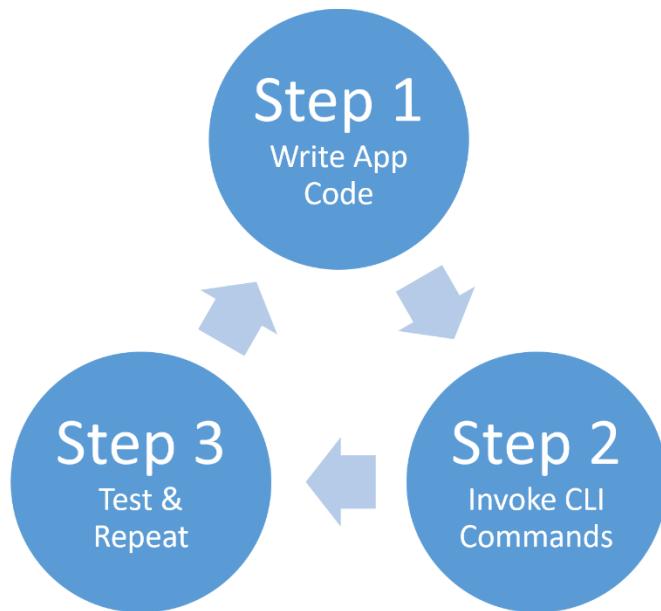


Figure 2.10 There are three steps to the NativeScript app development process: writing app code, using the CLI to create native apps, and testing the app.

2.3.1 Building NativeScript apps

You already know that NativeScript apps are written in XML, JavaScript, and CSS. This is the first step in your development workflow. Let's take a closer look at step 2, the NativeScript CLI. In step 2, the NativeScript CLI transforms your code into a native app, deploys the app to a mobile device (or emulator/simulator), and runs the app. You've already seen the CLI in action when we created and ran your first app: hello world.

When you created the hello world app, you used the `tns run <platform> --emulator` command to build, deploy, and run the app in an emulator/simulator. However, you may want to build your app without deploying and running it. You can build your app for either Android or iOS using one of the following `tns build` commands:

```
tns build ios  
tns build android
```

The `tns build` command packages your app code, the NativeScript runtime, and core modules into the native app (figure 2.11).

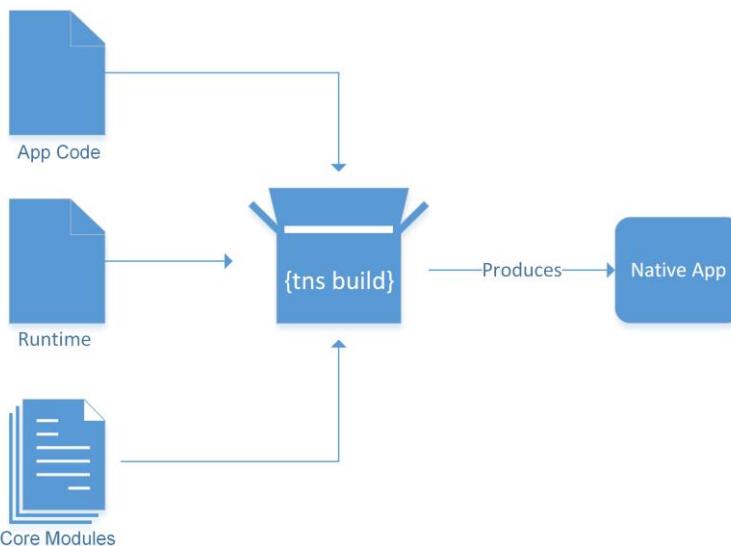


Figure 2.11 The `tns build` command bundles your app code, the NativeScript runtime, and the core modules into a native app

Every time that you build your app, whether it is through the `tns build` or `tns run` command, the NativeScript CLI takes your app code, along with the NativeScript runtime and core modules, and packages it into a native app for Android or iOS. At this point you may be thinking that this seems like a lot of work and that you are wasting a lot of time building, running, waiting for the emulator to launch, and so on. If you were thinking that, you are correct: it can take a lot of time, especially if you have a large app. So, let me introduce you to a better way: `livesync`.

2.3.2 Livesync

Because it's time-consuming to build, deploy, and re-launch your app every time you want to test a change, the NativeScript CLI quickly injects changes to your app code into a running app. This process is called `livesync`, and it's done automatically.

NOTE In earlier versions of NativeScript, you had to use a special CLI command to use `livesync`. But, it's now been fully integrated into the CLI, and you don't need to do anything special. Use `tns run`, and the code running on your app will be kept up-to-date automatically. Even though `livesync` is built-in, we still refer to the sync technology as `livesync`.

When you run the `tns run` command, the livesync engine NativeScript CLI calculates the delta file changes and synchronizes the files to the device (or emulator/simulator). Figure 2.12 shows how livesync works by replacing individual files within a running NativeScript app.

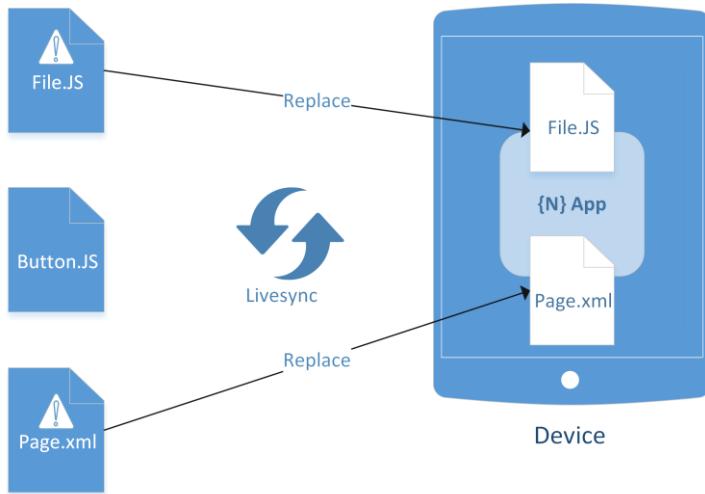


Figure 2.12 Livesync monitors your app's files for changes. When a change is found, it is synced to the running app.

2.4 Quick reference

We've touched on a lot of concepts in this chapter, and if you're new to NativeScript, you may be feeling overwhelmed. There's CLI commands to understand, cross-platform compilation, and syncing source code changes to running apps. Because this can be confusing, we've done two things for you. First, check out our NativeScript CLI quick reference guide in appendix B: it lists out the CLI commands you'll be using throughout the book. Second, is a getting started quick reference in table 2.2. It's got a recap of the NativeScript CLI commands you'll need use when creating a new app and running in an emulator/simulator.

Table 2.2 Getting started with the NativeScript CLI: the three commands you need to know

CLI Command	Description
<code>tns create <app-name></code>	Creates a new cross-platform NativeScript app named <app-name>. A folder will be created with the name of your app, and the NativeScript app structure described in chapter 3 will be added. This command also creates a vanilla NativeScript app, using JavaScript. For additional options, check out appendix B.

```
tns platform add <platform>
```

Adds the Android and iOS platforms to your app. This command is optional, because `tns run` executes this before building and deploying your app.

```
tns run <platform>
```

When your app has been created, take the shortcut and use `tns run android` or `tns run ios`. This will add the native platform, copy your app's source code into the native project, compile your app, install it on a device, emulator, or simulator, run it, then monitor it for changes using `livesync`. One command to rule them all. We like it.

2.5 Summary

In this chapter, you learned the following:

- Establishing a development workflow early on will allow you to focus on the creation of your app versus focusing on the different platforms.
- The NativeScript CLI is an integral part of your development workflow.
- How to use the NativeScript CLI to run various commands (`create`, `platform add/remove`, `run`).
- When you run an app with the `tns run` command, it monitors your app's source code for changes and syncs them to a running device automatically.

2.6 Exercise

In this chapter, we created a hello world app with a button and label. Try modifying the hello world app we created in this chapter in the following ways:

1. Create a new project using the “blank” project template
2. Run the help command for `run`
3. Build the blank project you created in exercise 1
4. Run the blank project you created in exercise 1 in an emulator

2.7 Solutions

1. `tns create ablankproject --template tns-template-blank`
2. `tns help run`
3. Either command:
 - a. `tns build ios`
 - b. `tns build android`
4. Either command:
 - a. `tns run ios --emulator`

b. tns run android -emulator

4-Jan-2019

Part 2: Structuring Your App

3

Anatomy of a NativeScript app

This chapter covers

- How NativeScript apps are structured
- How the NativeScript runtime loads the first page of your app
- How you can target different mobile platforms using conventions
- How to organize your NativeScript app code

In Chapter 2 you learned how to create your first cross-platform app with the NativeScript CLI, how to further use the CLI to add the Android and iOS platforms to an app, and how to test your app by running it in an emulator. In this chapter, we'll take a deeper look at the app structure the NativeScript CLI scaffolds.

The structure of a NativeScript app is important because NativeScript is an *opinionated framework* for developing mobile apps. Opinionated frameworks require you to write code, name files, and organize app components in certain ways; if you don't follow the rules of NativeScript, your app won't run. At this point, working with an opinionated framework probably sounds painful and frustrating. After all, why would you want to be restricted to following such specific rules? Don't worry: I've got your back. NativeScript's rules (also known as *conventions*) are easy to understand and make writing apps that work on Android and iOS simpler, not more complex. Let's jump in and learn!

3.1 Exploring the structure of a NativeScript app

All NativeScript apps have a common structure, automatically generated when you create your app with the CLI command `tns create <app name>`. For example, to create a new app named `myapp`, run the `tns create myapp` command. Figure 3.1, show the resulting scaffolded app structure.

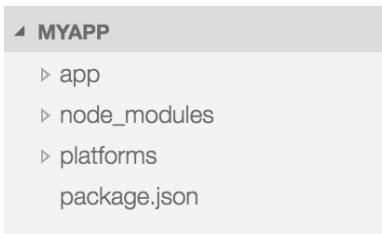


Figure 3.1 Running the `create` CLI command scaffolds the structure of an entire NativeScript app for you. Within the app's root folder are several files, a `node_modules` folder, and a folder named `app`.

TIP As we explore the structure of a NativeScript app, keep in mind that some files and folders should be left alone. You should not directly modify these files and folders. At first, this may be confusing and you may not understand which files and folders shouldn't be modified directly, so let's establish a ground rule.

Do not manually change files or folders unless they exist beneath the folder named `app`. The `app` folder is your playground. All other files and folders should be considered off limits (for now). As you continue through the book, we'll provide you with exceptions to this rule. The exceptions are truly exceptions, and we'll be sure to call them out to you, explaining why I've asked you to break this ground rule and make an exception. If you learn and follow this ground rule, you'll have an easier time learning NativeScript. To recap, if it's in the `app` folder, you can change it; otherwise, hands off!

3.1.1 The root application folder and files

The NativeScript app structure has several components organized in a tree-like hierarchy of files and folders (figure 3.2). At the root is a single folder, with a name of your app, as specified when running the `tns create <app name>` command. All files and folders contained within this folder are used by you and the NativeScript CLI to create a NativeScript app. Figure 3.2 shows the file and folder structure of a NativeScript app created with the `tns create myapp` command.



Figure 3.2 The file and folder organization of a newly-created NativeScript app. The `package.json` file is also pointed out as the only file in the root of the app's folder.

The `package.json` file (listing 3.1) is the sole file within the root app folder and is used to describe characteristics of your app and dependencies your app relies on. Because the file is simply a JSON-formatted text file, it's easy to view and edit.

NOTE Although the `package.json` file is located outside of the app folder, you can manually edit the contents. This is an exception to our ground rule.

Let's take a closer look at the default `package.json` file to understand the purpose of each portion.

Listing 3.1 package.json file contents

```
{  
  "description": "NativeScript Application", //A  
  "license": "SEE LICENSE IN <your-license-filename>", //B  
  "readme": "NativeScript Application", //C  
  "repository": "<fill-your-repository-here>", //D  
  "nativescript": {  
    "id": "org.nativescript.myapp" //E  
  },  
  "dependencies": {  
    "nativescript-theme-core": "~1.0.2", //F  
    "tns-core-modules": "3.1.0" //F  
  } //F  
}  
  
#A Provides a brief description of your app, features, and purpose  
#B Points collaborating developers to your license file, to describe what rights others have to contribute to,  
modify, alter, and redistribute your app code (optional)  
#C Points others to your app's README file  
#D The location of your app public or private code repository (optional)  
#E A NativeScript-specific section with an identifier for your app, used by Android and iOS platform to uniquely  
identify your app  
#F Listing of external libraries and library versions your app depends on, used by npm
```

You're likely familiar with the `package.json` file, which is used by npm and by npm modules maintained and installed with npm to describe the bundled code and to identify what dependencies this package relies on. Although NativeScript apps are not technically npm modules, NativeScript has adopted the `package.json` file to fill a similar role to that of the `package.json` file for npm modules. This file is central to your NativeScript project because it is the authoritative source for describing your app. Collectively, the `description`, `license`, `readme`, and `repository` fields provide others with information they may find helpful when reviewing your code or collaborating with you. Although these first four fields provide others with contextual information about your app, the `nativescript` and `dependencies` sections are much more important.

NATIVESCRIPT SECTION

The `nativescript` section has a key field: `id`. The `id` field contains a unique identifier formatted in *reverse domain name notation*. This unique identifier is used by NativeScript when building your app and is copied from this location in the `package.json` directly into the native app files for Android and iOS.

DEFINITION Reverse domain name notation is similar to a website's domain name. Take <http://brosteins.com> as an example. Brosteins.com is the domain name, and com.brosteins is the reversed domain name notation.

What's important to note is that every app you create should have a unique identifier. There are no official rules preventing you from using any unique identifier (other than being unique), but it's recommended you use a unique identifier that is both meaningful to your app and contains a related domain name you're authorized to consume. For example, we created a NativeScript app for my son called *My Robot* and used the unique identifier of com.brosteins.myrobot (listing 3.2).

Listing 3.2 The Brosteins My Robot app unique identifier, as found in the package.json file

```
{
  ...
  "nativescript": {
    "id": "com.brosteins.myrobot"      //#
  }                                     //#
  ...
}

#A We chose the reverse domain name of com.brosteins.myrobot because we own the domain name and the
app's name is My Robot
```

DEPENDENCIES SECTION

The second section of the *package.json* file of importance is the *dependencies* section. This section prescribes to npm module specifications, meaning npm uses it to identify the name of other npm modules used by your app. When you create your app, the *nativescript-theme-core* and *tns-core-modules* modules is added as a dependency (listing 3.3).

Listing 3.3 tns-code-modules dependency included in package.json

```
{
  ...
  "dependencies": {
    "nativescript-theme-core": "~1.0.2",
    "tns-core-modules": "3.1.0"
  }
}
```

as 4-Jan-2019 it is v5.1.0

NOTE Don't worry if you don't know what the *nativescript-theme-core* and *tns-core-modules* modules are for: you'll be learning about them throughout the book. For now, you need to know that they're needed by NativeScript to make an app.

3.1.2 The node_modules folder

The first folder underneath the root folder of your app that we'll investigate is the *node_modules* folder. This folder is closely related to the *package.json* file we just looked at, because it contains a local copy of your app's npm package dependencies, as outlined in the *package.json* file.

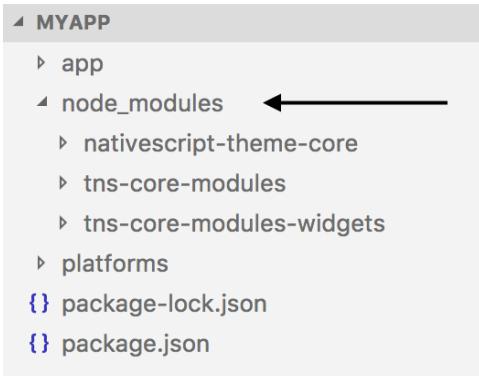


Figure 3.3 `node_modules` folder containing a local copy of the packages needed, as defined by the `package.json` file's dependencies section.

During your app development lifecycle, the `node_modules` folder is a folder you'll see a lot; but you will not interact with it directly.

NOTE Throughout the book, I'll be using `npm` to add new packages, which will cause the contents of the `node_modules` folder to change. This is ok, because I'm not changing the contents of the folder directly.

NOTE You may notice that `tns-core-modules-widgets` module also included in the `node_modules` folder. This is a dependency of the `tns-core-modules` module and is not listed in the `package.json` file explicitly.

3.1.3 The `platforms` folder

The second folder underneath the root folder of your app is the `platforms` folder. The `platforms` folder is critical to your NativeScript app; but, when you create an app, the folder is empty.

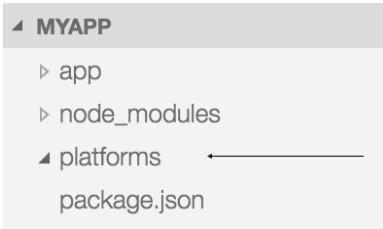


Figure 3.4 An empty `platforms` folder, as seen immediately after creating your app.

The `platforms` folder is critical to your app because it houses platform-specific files, folders, and resources needed to build native Android and iOS apps. Although the `platforms` folder is created when an app is created (using the `tns create <app name>` CLI command), it is not used until you add a native

platform to an app using the `tns platform add ios` or `tns platform add android` CLI commands. When the `tns platform add <ios or android>` command runs, an *android* or *ios* **subfolder** is created within the *platforms* folder. Figures 3.5 and 3.6 show running the `tns platform add <ios or android>` command and the resulting subfolders created within the *platforms* folder.

```
[Michaels-MBP:myapp mike$ tns platform add ios
Copying template files...
Project successfully created.
[Michaels-MBP:myapp mike$ tns platform add android
```

Figure 3.5 Running `tns platform add <platform>` commands for ios and android from the command prompt.

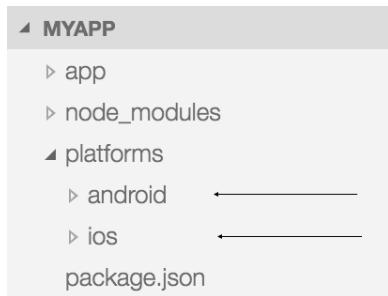


Figure 3.6 Platform-specific subfolders for Android and iOS created within the platforms folder after running the `tns platform add <android or ios>` CLI commands.

Like the `node_modules` folder, the `platforms` folder is automatically generated and maintained for you by NativeScript.

NOTE Generally, the `platforms` folder is hands-off, but future chapters will introduce you to scenarios where you may need to modify files in the `platforms` folder.

3.1.4 The app folder

Finally, the `app` folder. Up until now, we've been looking closely at all the supporting files and folders that make up a NativeScript app, but, we haven't looked at the code you will write.

You'll recall from chapter 1 that NativeScript apps are composed of user interface markup written in XML, styled with CSS, and supported with business logic written in JavaScript. But where do these files go?

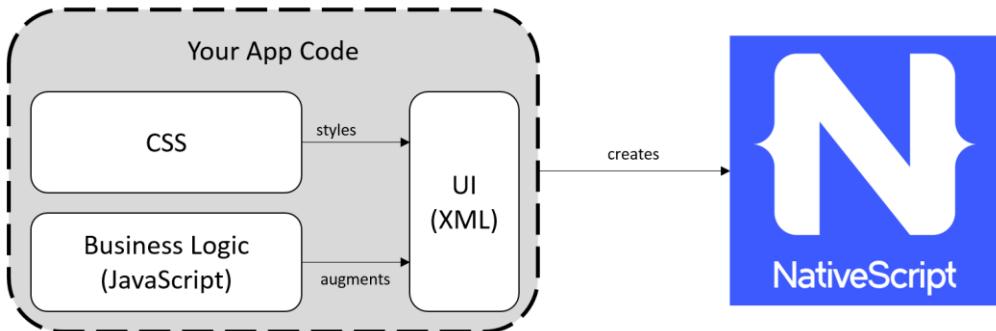


Figure 3.7 The code components of a NativeScript app are XML user interface code, CSS styling code, and JavaScript business logic code.

The `app` folder is where you'll spend most of your time developing NativeScript apps because it contains all of your XML, CSS, and JavaScript code. In fact, aside from some minor edits to the `package.json` file, we usually forget about files and folders outside of the `app` folder while I'm developing and testing my apps.

Inside of the `app` folder you'll find the `App_Resources` folder, several files named `app.<extension>`, `bundle-config.js`, `main-page.<extension>`, `main-view-model.js`, another `package.json` file, and a file named `references.d.ts`.

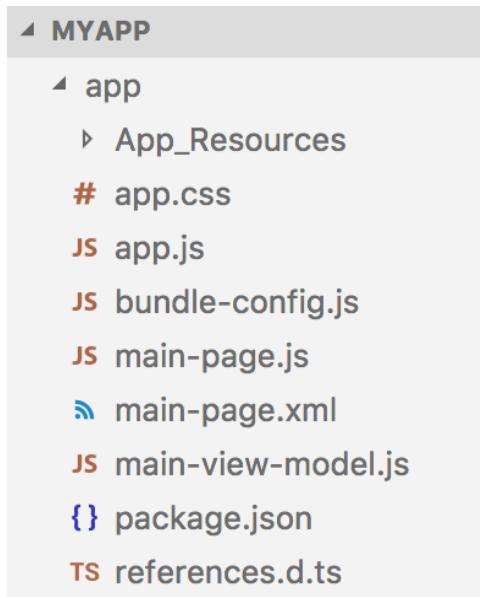


Figure 3.8 The `app` folder contains a folder named `App_Resources`, several files named `app.js`, `app.css`, `main-`

page.xml, main-page.js, main-view-model.js, package.json, and references.d.ts.

Let's start by looking at the `App_Resources` folder.

APP_RESOURCES

Despite attempts to merge mobile development platform capabilities into a single set of common commands, APIs, and user interface elements, no cross-platform mobile environment will bridge the gaps between every mobile platform 100%. NativeScript gets close, but there are some aspects of Android and iOS development that are fundamentally different. For example, screen resolution and dots per inch (DPI) will vary greatly between hardware devices running Android and iOS.

DEFINITION Screen resolution is a measure of the number of pixels on a screen, usually described in the form of width x height. For example, a screen resolution of 640x480 means the screen is 640 pixels wide and 480 pixels high.

DEFINITION Dots per inch (DPI) is a measure of dot density, and is usually used in the printing industry to describe the number of printed dots appearing in a square inch of a printed book or magazine. When referring to screens, the concept of a "dot" is often confused with a "pixel." Screens have pixels, not dots; therefore, their density is measured in pixels with pixels per inch (PPI). Although DPI and PPI are technically different, most people don't differentiate between the two. In fact, the Android platform prefers the terminology of DPI versus PPI. Through this book, we will use the term DPI.

NativeScript compensates for the gaps between Android and iOS by placing platform-specific customizations in the `App_Resources` folder. Beneath the folder you'll find platform-specific folders for Android and iOS.

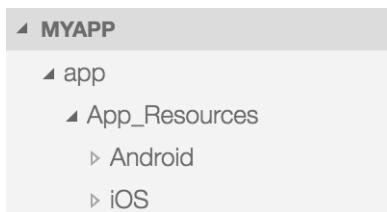


Figure 3.9 File system hierarchy showing an android and an ios sub folder beneath the `App_Resources` folder.

The platform-specific folders contain files that only their platform cares about: the `Android` folder has files that the Android platform needs, and the `iOS` folder contains files the iOS platform needs. It's not important for you to know what's inside of the `Android` and `iOS` folders right now. Later in this chapter, and throughout the book, I'll explain the contents of the `App_Resources` folder in more detail.

One detail that is important to call attention to is the similarity between the `platforms` and `App_Resources` folders. As you'll recall from earlier in this chapter, the `platforms` folder also contains platform-specific files and folders. In a way, the two folders are similar, but they are also different for

many reasons. The most important difference between the two folders is you *will* make direct and manual changes to files in the *App_Resources* folder, but not the *platforms* folder.

NOTE Remember the ground rule: it is safe to edit files and folders beneath the *app* folder. Because *App_Resource* is beneath the *app* folder, it is safe to change.

~~It's not important that you understand what changes you will be making right now, but it is important to understand how the platform-specific files within *App_Resources* contribute to an app.~~

After you've made platform-specific changes to files under the *App_Resources* folder, you will run the `tns prepare <android or ios>` CLI command. This command reads the platform-specific changes from the *App_Resources* folder, then merges the changes into the native project files in the *platforms* folder (figure 3.10).

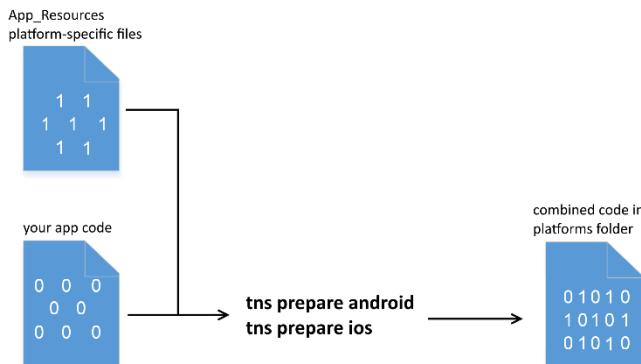


Figure 3.10 When you run the `tns prepare ios/android` command, the CLI copies and merges platform-specific files into the corresponding native platform folders in the *platforms* folder.

REFERENCES.D.TS

The *references.d.ts* file is a special file used by some integrated development environments (IDEs) and editors to provide *IntelliSense*-like functionality for you.

DEFINITION *IntelliSense* is a technology built into many IDEs and text editors providing automated suggestions for “code completion” when typing code. The suggestions are typically surfaced to you through pop-ups next to or near the text you are typing.

Technically, this file has nothing to do with your NativeScript app, and only affects you if you were writing your NativeScript app in TypeScript instead of JavaScript. We won’t go into detail about the *reference.d.ts* file (or its contents). For now, just ignore the *references.d.ts* file. But, if you’re curious, just can’t ignore it, or are interested in learning more about TypeScript, check out our summary of TypeScript later in this chapter.

DEFINITION TypeScript is a programming language, specifically a superset of JavaScript, adding strongly-typed and class-based object-oriented capabilities. JavaScript virtual machines cannot natively understand TypeScript, so TypeScript is transpiled (or converted) to JavaScript before being run.

BUNDLE-CONFIG.JS

This file is used to configure webpack (if it's installed into your app).

DEFINITION Webpack is a module bundler, meaning that it can take various JavaScript modules spread across multiple files and bundle (or pack) them into a single, compressed, and optimized format.

If you're a web developer, you may have heard of webpack. We feel it's important to point out the `bundle-config.js` file, and that you can use webpack with NativeScript apps to make them smaller, more efficient, and load faster. But, that's as far as we're going to go. Configuring and using webpack in a NativeScript app is a more advanced topic that we're not going to cover. If you're interested in learning more, checkout <https://www.nativescript.org/blog/improving-app-startup-time-on-android-with-webpack-v8-heap-snapshot>.

MAIN-VIEW-MODEL.JS

The second file we'll examine in the `app` folder is the `main-view-model.js` file. As you can tell by the file name extension, this file is a JavaScript file. In NativeScript, JavaScript files typically contain application and business logic code. At this point in the book it's a little too early to take a deep dive into the contents and functionality of the `main-view-model.js` file, but don't worry because we will be looking at it in a later chapter. For now, it's only important to know it contains application code used by the next two files we'll examine: the `main-page.xml` and `main-page.js` files.

MAIN-PAGE.XML AND MAIN-PAGE.JS

The third set of files within the `app` folder are the `main-page.xml` and `main-page.js`. As you can tell by the file names, `main-page.xml` is an XML file describing the user interface of a page within the app, and `main-page.js` contains the page's corresponding business logic. Together, these two pages form a cohesive unit, representing a single page, named `main-page` (because of the file names). Later in this chapter, you'll learn more about the concept of a page and how files named in a similar manner are important. Right now, you can think of a NativeScript page as something similar to an HTML page.

PACKAGE.JSON

The fourth file in the `app` folder is another `package.json` file. This `package.json` file is similar to the `package.json` file found in the root of the NativeScript app. You'll recall that the root `package.json` file is used to describe the overall app and dependencies required for your app to build and run. This `package.json` file is similar but instead of describing the overall app, it describes the configuration and contents of the `app` folder.

NOTE You may find it strange that a `package.json` file exists within the `app` folder. After all, you have already seen a `package.json` file in the app's root folder; but there's a reason behind its existence: the contents of the `app` folder is actually an npm package. When npm packages are created, the author is

required to include a `package.json` file describing the package. As a result, because the contents of the `app` folder comes from an npm package, it contains a `package.json` file. Mystery solved.

Earlier in this chapter, you saw how the CLI's `tns create` command scaffolds the file and folder structure of a NativeScript app, but if you were to pause the CLI in the middle of the scaffolding process, you would notice the `app` folder is empty. That's because the contents of the `app` folder are dynamically added from an npm package during app creation. More specifically, when the `tns create` command runs, the CLI silently runs npm and installs the `tns-template-hello-world` npm package into a temporary directory and merges the contents of the package into the app folder.

{tns create myapp}

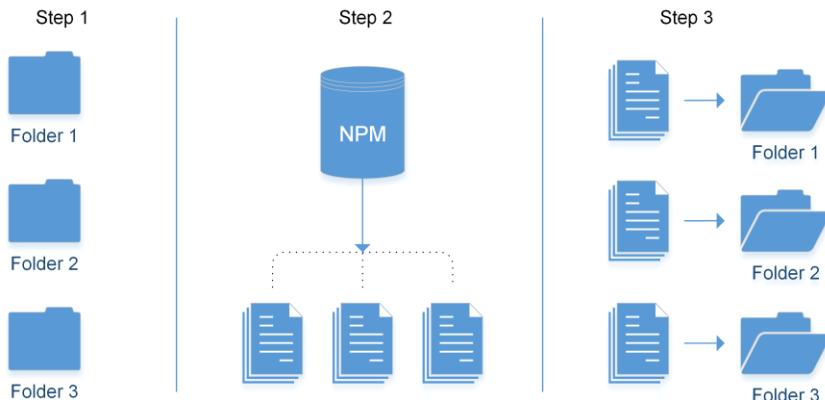


Figure 3.11 When you run the `tns create myapp` CLI command, empty folders are created (step 1), the `tns-template-hello-world` npm package is downloaded (step 2), and the contents of the npm package is merged into the empty folders (step 3).

Because NativeScript is open source and developed on GitHub, you can publicly see the contents of the "Hello World" template, located at <https://github.com/NativeScript/template-hello-world>. When you browse out to this URL, you'll see that the contents of the repository look identical to the scaffolded `app` folder.

Although the contents of the `app` folder came from an npm package, which requires that a `package.json` file exists, NativeScript uses the file for more than a description of the original npm package. Let's take a closer look at the contents of the `package.json` file and differentiate which values exist due to npm's requirements and which are true NativeScript configuration values. You should take special note of the NativeScript configuration values because we will be revisiting them in later chapters in greater detail.

Listing 3.4 Contents of the app\package.json file

```
{
  "android": { //#A
```

```

    "v8Flags": "--expose_gc" //#A
  }, //#A
  "main": "app.js", //#B
  "name": "tns-template-hello-world",
  "version": "3.1.0" //#C
}
#A Okay to ignore this setting, but if you're interested, it is NativeScript setting to configure how the V8
JavaScript virtual machine is run on Android. Don't change it!
#B NativeScript setting for the application's main entry point. Take note of this setting!
#C Values needed by npm; can be ignored because you don't need to alter them

```

In the previous listing, you can see that many of the key/value pairs within the `package.json` file are specific to npm. You can ignore these pairs. The `android` pair is a NativeScript-specific setting, but they're used internally by NativeScript and shouldn't be changed. You can also ignore it. In fact, there is only one pair of real importance to you: `main`, with a value of `app.js`.

The `main` key/value pair is used by NativeScript to identify your app's main point of entry. By default, this is set to `app.js`, telling the NativeScript runtime to load and execute the code within the `app.js` file when the app starts.

APP.CSS AND APP.JS

The fifth and final set of files residing in the `app` folder are the `app.css` and `app.js` files. The `app.css` file contains global UI styling, which is loaded by the NativeScript runtime when your app starts. You'll learn about UI styling in a future chapter. For now, remember the `app.css` file contains global styles and is loaded when an app runs.

The `app.js` file is the first code that runs when an app starts. NativeScript knows to bootstrap the `app.js` file because of the `main` setting of the `package.json` file (figure 3.12).

```

{} package.json ✘
1  {
2    "android": {
3      "v8Flags": "--expose_gc"
4    },
5    "main": "app.js", ←
6    "name": "tns-template-hello-world",

```

Figure 3.12 When your app runs, the NativeScript runtime bootstraps your code by reading the value of the `main` name/value pair from the `package.json` file in the `app` folder. By default, `main`'s value points to the `app.js` file. The `app.js` file is then run.

DEFINITION Bootstrapping is the act of loading a minimal amount of initialization code into a system to facilitate the further loading and execution of the main application code. In many systems, the bootstrapping process points to a specific place in memory, line of code, or function name to be run first.

NOTE In NativeScript, your code is bootstrapped by specifying the main code file containing your code, which defaults to the *app.js* file.

Saying that the *app.js* file is run during the bootstrapping process isn't enough, because it launches the app code that you write. You may be asking what happens inside of the *app.js* file to load and display the "home page" of your app? Let's look at the *app.js* file for the answer.

```
var application = require("application");
application.start({ moduleName: "main-page" });
```

The *app.js* file has only two lines of code inside. The first line loads and obtains a reference to the *application module*. In short, the *application module* is a collection of methods and objects that allow you to control global app behaviors, such as starting the app. You'll be learning about NativeScript modules in a later chapter, so it's ok if you don't fully understand the first line of code yet. The second line of code tells NativeScript to start your app by loading and displaying a *module* named *main-page*. We like to think of this module as your app's home page because it's the first user interface page you'll see in your app. Every app will have an *app.js* file with (at least) these two lines of code – without them, your app will not load and display a home page.

3.2 Understanding app startup

As you have learned about the basic structure of a NativeScript app, you've seen individual components that cause your app to startup and load the first page of your app. Knowing about the individual components is ok, but let's look at how all of these components work together to orchestrate app startup.

To understand NativeScript app startup, let's look at something you're familiar with: an HTML application. Imagine you've created an HTML application that you want to host at <http://brosteins.com>. Your application contains one file: *index.html*. When you publish your application to your web server, you configure the default home page for <http://brosteins.com> to be the *index.html* file, so users will automatically see *index.html* when they browse to <http://brosteins.com>.

NativeScript apps work just like your HTML application – you place your app files into the *app* folder, configure a starting page, and when you run the app, the NativeScript runtime reads your configuration and loads the home page of your app (figure 3.13).

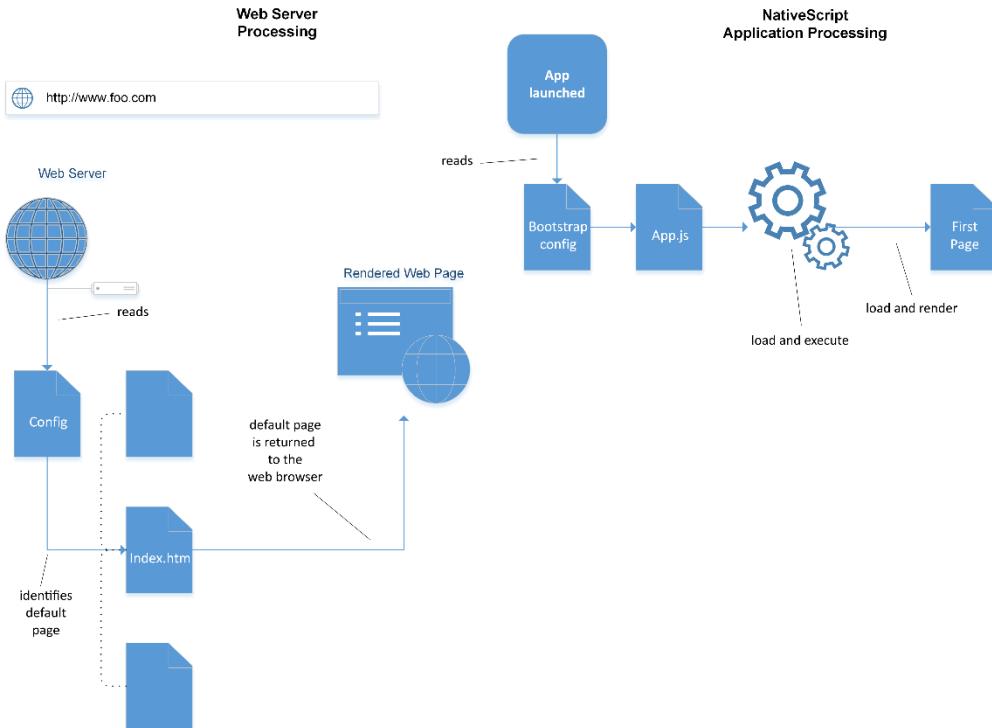


Figure 3.13 When you type a URL, the webserver reads its home page configuration to determine which page to return (the default page). The default page is then returned to the web browser and rendered as a web page. In NativeScript when the app starts, its bootstrap configuration setting is read (which by default points to the app.js file), the app.js file is loaded and executed, and upon execution, the application module points to a home (first) page. The home (first) page is then loaded and rendered to you.

In an HTML application, when you point your web browser to the application's URL, your web server reads its configuration to determine the home page (`index.html`) and returns the home page to your web browser. When a NativeScript app is run, the NativeScript runtime reads its configuration from the `package.json` file, looking at the `main` key/value pair to determine the file to use as the main bootstrap code. By default, NativeScript bootstraps the `app.js` file, executing code that tells your app to start and load a page (which is `main-page`).

As you've been learning how NativeScript apps load your home page, you may have noticed that the home page configuration points to a page called `main-page`, which is not actually a file in the app directory. `Main-page.xml` and `main-page.js` are files, but what's the deal with `main-page` without an extension? When you tell NativeScript to look for `main-page` (without an extension), it automatically assumes there is a file named `main-page.xml`.

Let's look at the second line of the `app.js` file again:

```
application.start({ moduleName: "main-page" });
```

The application is started by pointing to the page named `main-page`, which is assumed to be a file named `main-page.xml`. You previously learned how similarly named files, like `main-page.xml` and `main-page.js`, together form a cohesive unit. For now, let's call this cohesive unit a *page*.

DEFINITION A *page* is a collection of similarly-named XML, CSS, and JavaScript files that can be referenced by ignoring the filename extension. For example, `main-page.xml`, `main-page.css`, and `main-page.js` can collectively be referred to as a page named `main-page`.

When your NativeScript app runs and is told to load a page named `main-page`, the NativeScript runtime knows (by convention) to search for three files: an XML file, a CSS file, and a JavaScript file named `main-page`. Figure 3.14 shows how the NativeScript runtime uses this file-naming convention to represent a page.



Figure 3.14 When the `main-page` page is referenced, the NativeScript runtime looks for and loads files named `main-page.xml`, `main-page.css`, and `main-page.js`.

The similar names of user interface and business logic file isn't by mistake or by chance. In fact, we've stumbled upon a foundational tenet of NativeScript: conventions. This file-naming convention is key to how NativeScript works and is an important concept to learn early on.

MORE CONVENTIONS

There are various other conventions in NativeScript that make it easy to use, but jumping into them right now will distract you from what's important: learning NativeScript. We'll be touching on the conventions throughout the book, but we've also put them in appendix C. If you'd like to learn about them now, feel free to break away from chapter 3 and return here when you're finished.

3.3 Style guide and app organization

Earlier in this chapter, you learned about the structure of a NativeScript app, and how the app folder contains a collection of your app's user interface (XML), styling (CSS), and business logic (JavaScript) code.

You also learned that similarly-named files (`main-page.xml`, `main-page.css`, and `main-page.js`, for example) are treated as a single unit, which we have called a *page*.

What you haven't learned is that NativeScript apps are built from a collection of pages. When an app is created with the NativeScript CLI, the `main-page` page is placed in to root of the `app` folder. There is nothing special about this location, except that your `app.js` file points to the root of the `app` folder to initially load `main-page`:

```
var application = require("application");
application.start({ moduleName: "main-page" });
```

As your app grows, you may start by placing additional pages in the root of the `app` folder, but this can quickly become overwhelming. Imagine an app with 25 distinct pages, each having an XML, CSS, and JavaScript file. 75 files are a lot of files to scroll through. We'll argue that it's too many to scroll through; so you need to think about organizing your app's pages right from the beginning.

Although there's no right or wrong way to organize the pages within your app, we'll share some helpful tips to make your NativeScript app more manageable.

TIP Group each set of page files in a folder with the same name as the page.

Let's assume your app has a `contact-us` page (figure 3.15).

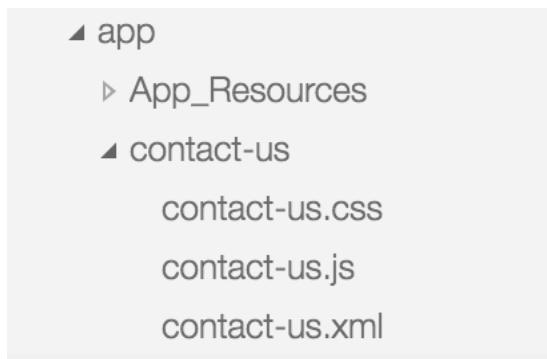


Figure 3.15 Placing the `contact-us.xml`, `contact-us.css`, and `contact.js` files together in a folder named `contact-us` helps to organize pages within your app.

You can see that the `contact-us` page files (`contact-us.xml`, `contact-us.css`, and `contact-us.js`) are located inside of a folder named `contact-us`. This page organization helps keep similar and related files together. Although this organizational technique is used to make it easier to find, locate, and manage your app's files, it makes a huge difference and can save you from scrolling through hundreds of files in the root of the `app` folder.

TIP Group related pages together by feature or functional area by placing related page folders into another folder. The structure/form of your app pages and files should match the functionality of your app; in software development, this is often referred to as "form follows function."

Let's use the Phone app on iOS as an example to illustrate how the structure of your app pages should follow their functionality. Figures 3.16 and 3.17 show how form can follow function.

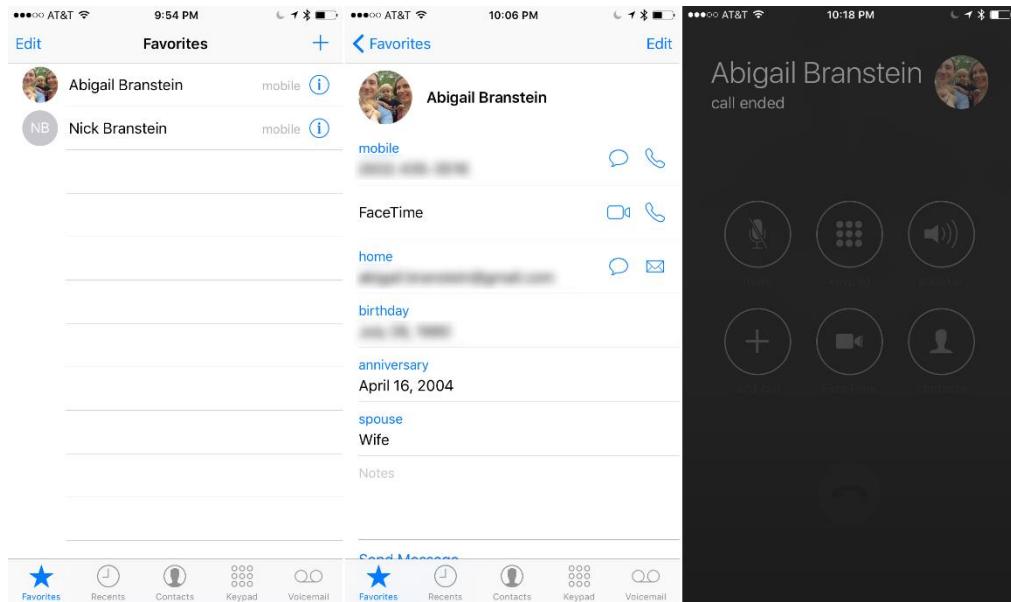


Figure 3.16 iOS's Phone app starts on the Favorites tab, showing your Favorite contacts. You can navigate to the contact details by tapping the information icon, or call a favorite by tapping their name.

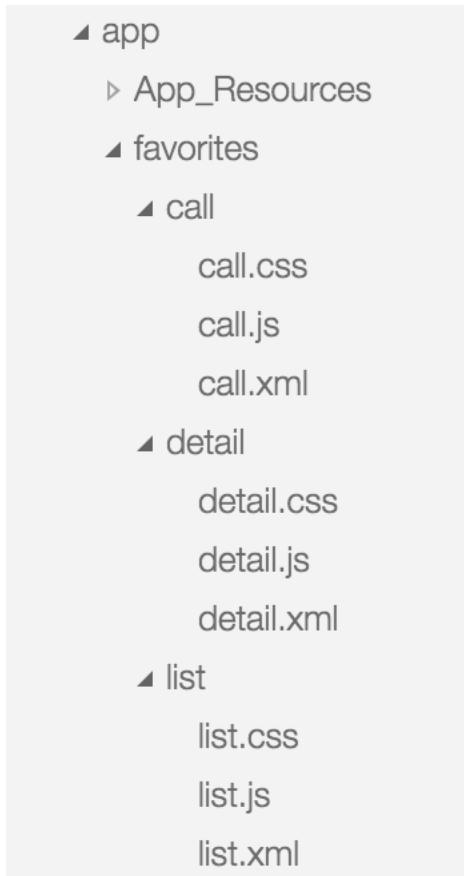


Figure 3.17 Based on the Phone app's functionality, you could create an overarching favorites folder, containing sub-folders for each of the three pages: list, call, and detail.

You will see how the Favorites tab within the Phone app displays a list of favorite contacts. From this list, you can tap a contact to call them, or tap the information icon and navigate to a contact details page. If you were designing this app in NativeScript, you should think about how to logically organize these three distinct pages. Because the pages are related to the Favorites functionality, you could create an overarching *favorites* folder, containing the page-level folders for the main favorites list (the *list* folder), calling a favorite (the *call* folder), and viewing the contact's details (the *detail* folder).

TIP Be agile.

Our final recommendation is to be agile. Don't get tied into a specific way of organizing your app's pages, never intending to change them. In fact, you should change how your apps are organized as often as their functionality changes and dictates. Remember what's important. Don't just create an amazing

app, but create an app that is easy to maintain and easy for others to understand by looking at the structure and code. Don't spend an exorbitant amount of time organizing your app, but then again, don't dismiss the organization early on. If you make your app's structure visually appealing, you'll thank yourself in the long run.

Now that you've learned how to structure your app and organize pages into folder, you'll continue to learn about pages in the next chapter. You'll learn how to structure the user interface code within a page, and how to navigate between pages.

3.4 Summary

In this chapter you learned that:

- All NativeScript apps have folders named *platforms*, *node_modules*, and *app* folder.
- You shouldn't directly modify the contents of the *node_modules* and *platforms* folders.
- The *app* folder contains all your app's code that you develop.
- Pages consist of similarly-named user interface (XML), style (CSS), and business logic (JavaScript) code files.
- When organizing your app's code, page files should be placed into a folder with the same name as the page.

3.5 Exercise

In this chapter, you learned how NativeScript apps are organized. Try using what you learned to do the following:

- Change the starting page of your app to a new page named *home-page*.

3.6 Solutions

To change the starting page of an app to *home-page*:

- Create a new file named *home-page.xml*.
- Change the *app.js* file to launch the *home-page* page when the app loads: `application.start({ moduleName: "home-page" });`

04-Jan-2019

4-Jan-2019

4

Pages and navigation

This chapter covers

- Creating multi-page apps
- Adding multiple UI elements to each page
- Responding to events and navigating between app pages

In chapter 3 you learned how NativeScript apps are structured. You also learned how the conventions of NativeScript help keep your app code organized and make development easier. Now it is time to take what you have learned in the previous chapters and start building a professional mobile app.

Over the next several chapters, you'll be learning how to create a multi-page mobile app for Tekmo, an imaginary company that sells retro video games and video game accessories. Tekmo wants to broaden its reach to customers by creating a mobile app for iOS and Android that will showcase the products they sell. This mobile app is their first foray into the mobile space, and they want to keep it simple, mimicking their website. The app will consist of the following four pages:

- Home
- About

- Contact Us
- Products

In this chapter, we'll be building the first two pages of the Tekmo app: Home and About. The Home page will be the first page app users see when they launch the Tekmo app. From the Home page, users will be welcomed and can then navigate to the other three pages. The About page will share the history of the Tekmo company, their passion for gaming, and the company's mission statement. Through building these two pages, you'll learn how to create multi-page apps and how to navigate between pages. In future chapters, we'll finish the Tekmo app by building pages to submit questions and feedback to Tekmo (the Contact Us page) and browse a listing of the retro video games for sale (the Products page).

Let's get started!

4.1 Creating a multi-page app

In chapter 3, we discussed the organization of a NativeScript app; let's take that knowledge we learned in chapter 3 and apply it by creating a new multi-page NativeScript app. Most NativeScript apps are a collection of multiple pages, and you navigate between pages. If this sounds a little overwhelming right now, don't worry. NativeScript provides several easy ways to navigate between pages, and you'll be learning about them throughout the book! Now, let's dig in and learn about pages and navigation while we build the Tekmo app.

Start by firing up your favorite command-line tool and scaffold a new NativeScript using the command line, just as you did when you created the hello-world app in chapter 2. Use the `tns create` command name the app `Tekmo`.

```
tns create Tekmo --template tns-template-hello-world
```

In chapter 2, we discussed that we could have used the `template` parameter to create the Hello World app. The `template` option tells the CLI to use the template named `tns-template-hello-world`, which is a simple app template. We like to start off our projects by using a simple template because it provides a minimalistic starting point that requires nothing additional to start building an app.

Once the CLI has finished creating the Tekmo project, open the `Tekmo` folder in your editor of choice.

TIP Visual Studio Code (VS Code) is a free editor created by Microsoft, geared at providing a light-weight text-editing environment for developers. We love VS Code and consider it our editor of choice for developing NativeScript apps. We'll be using VS Code throughout the book. If you're trying to decide which editor to use, give VS Code a spin. You can download Visual Studio Code <https://code.visualstudio.com>.

NOTE Before you move on, we need to clean up the template and remove various defaults. Delete the `main-view-model.js` file, then delete the contents of the `main-page.js` file. Finally, replace the contents of the `main-page.xml` file with this markup: `<Page></Page>`.

In chapter 3, you learned some organizational techniques that you can use to help keep your app organized. Our app will be small, so we'll be using a single folder named `views` to organize all of our page.

Go ahead and create a new folder called `views` below the `app` folder. Your app files and folders should look like figure 4.1 after creating the `views` folder.

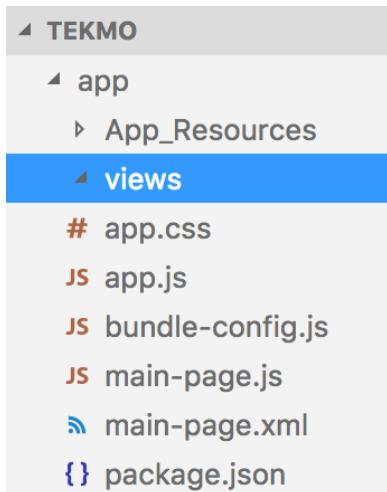


Figure 4.1 The resulting file and folder structure of the Tekmo app after scaffolding a new app and creating a `views` folder

Great start, now it's time to fill up the `views` folder with some pages.

4.1.1 *Creating the Home page*

The first page you create should be your app's main page, which is the page that is loaded when your app loads. You'll recall from chapter 3 that the `app.js` file contains the `application.start` code that tells NativeScript what view to load when the app launches (as seen in listing 4.1).

Listing 4.1 The `app.js` file of a new NativeScript app

```
require("./bundle-config");
var application = require("application");
application.start({ moduleName: "main-page" });
//A
#A The default home page for a NativeScript app is called main-page, which loads the main-page.xml file
```

By default, the `app.js` file loads the `main-page` page, which corresponds to the `main-page.xml` file. We'll update this in a moment, but let's quickly review NativeScript pages and then create our new home page.

Pages are a collection of three files: XML, CSS, and JavaScript, as shown in figure 4.2.

NativeScript construction of the “Main-Page”

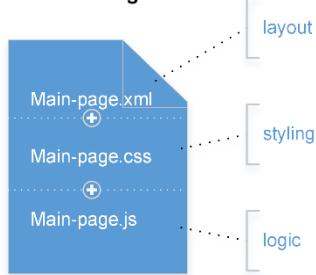


Figure 4.2 A page in NativeScript is composed of 3 components: XML, CSS, and JavaScript.

The XML file defines the UI, CSS provides a mechanism for styling the layout, and business logic that interacts with and collaborates with the UI resides in the JavaScript file.

Although NativeScript pages are defined as XML, CSS, and JavaScript, you don't need to create all three files; at a minimum, you need to create an XML file to define the UI elements of a page. The remaining files (CSS and JavaScript) are optional. If you include them, NativeScript will automatically load them when you open a page, but if they're not there, NativeScript won't complain and only load the XML file.

TIP When you're creating new pages, only create the files that you need. If you only need to define a UI, only create the XML file. As your page begins to take shape and evolve, you can always add the CSS and JavaScript files when they're needed. Starting with the XML file can save you time.

We're going to use our own tip and start with the UI of our Home page right now. Create the Home page of the Tekmo app by adding a new file named *home.xml* underneath the *views/home* folder (figure 4.3).

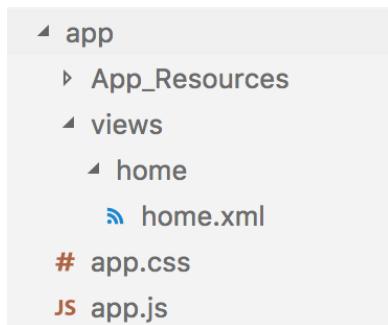


Figure 4.3 The *home.xml* view added to the *views/home* folder

The Home page that you added to the Tekmo app is obviously blank; you'll be adding UI elements to it soon, but let's tell our app to load this new page when the app loads. You'll recall that the `app.js` file is the entry point for every NativeScript application. Let's set the Home page to `home.xml` when the app is launched by updating the `app.js` file with the code in listing 4.2.

Listing 4.2 The app.js file updated to launch the home.xml file

```
var application = require("application");
application.cssFile = "./app.css";
application.start({ moduleName: "views/home/home" }); //##A
#A Update the starting module name to point to the newly created home page located in the views/home folder
```

NOTE Remember the conventions that we discussed in chapter 3? When referencing a module (in this case a page), NativeScript knows that this is an XML file so you do not need to put the filename extension on to `home.xml`. Pointing NativeScript to the `views/home/home` location is enough to load the page.

When your app loads, the code in the `app.js` file is executed. Let's verify this by running your app. Use the `tns run` CLI command in your command prompt to ensure everything is working correctly.

```
tns run ios --emulator
tns run android --emulator
```

You should see a blank screen when your app loads, as shown in figure 4.X.

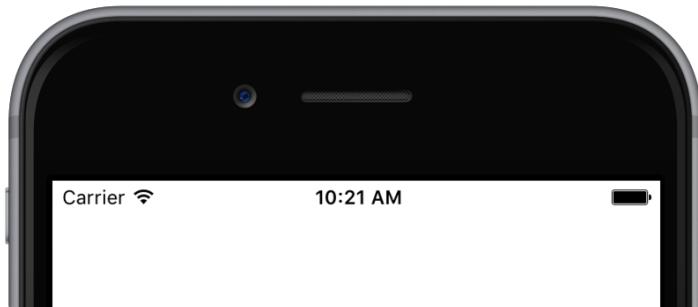


Figure 4.4 A blank home page is displayed after setting the new main page of the Tekmo app to `views/home/home`.

4.1.2 Adding content to the Home page

It is time to update the `home.xml` file and add some much-needed content to the Home page (we weren't going to have you leave it blank forever). Add a welcome message the page using the code in listing 4.3.

Listing 4.3 The views/home/home.xml file

```
<Page> //##A
<StackLayout> //##A
```

```
<Label text="Welcome to the Tekmo App!" /> //#B  
</StackLayout>  
</Page>
```

#A The **Page** element is the container for all other elements in a page and the **StackLayout** element tells the NativeScript runtime to place the elements that it renders on top of each other. You will learn more about the **StackLayout** in chapter 5

#B A label to display text on the screen

As you may have already noticed, the XML markup of your page looks like the HTML of a webpage. We'll explain each of the UI elements in just a minute. But before we do, we'd like you to see the parallel between NativeScript's XML and HTML you'd find in an HTML application. Figure 4.4 illustrates these similarities.

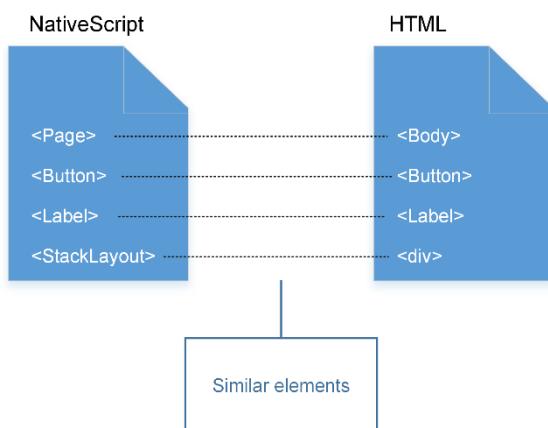


Figure 4.4 Comparison of the *home.xml* in NativeScript to the HTML of a similar webpage

Just because NativeScript pages are defined using XML doesn't mean that you can put any XML element into the page. As you continue through this book you'll learn about various XML elements that work together. If you just can't wait, you can find a complete reference of the available references online at <http://docs.nativescript.org/ui/components>.

Now, let's get back to building the Home page with the code in listing 4.4, which is the same code shown in listing 4.3, but shown here for convenience.

Listing 4.4 The views/home/home.xml file

```
<Page> //#A  
  <StackLayout> //#A  
    <Label text="Welcome to the Tekmo App!" /> //#B  
  </StackLayout>  
</Page>
```

#A The **Page** element is the container for all other elements in a page and the **StackLayout** element tells the NativeScript runtime to place the elements that it renders on top of each other. You will learn more about the **StackLayout** in chapter 5

#B A label to display text on the screen

All NativeScript XML pages must begin with the `Page` element. The `Page` element is the parent container for all other elements in a NativeScript page. Just like the `body` tag of a HTML page, you place all the page's content and other UI elements that you want to display on your NativeScript page inside of the `Page` element. Figure 4.5 shows this concept, with several UI elements nested underneath the `Page` element.



```
<Page>
  ...
  <Button />
  ...
  <Label />
  ...
</Page>
```

Figure 4.5 All elements that you want to render on a page in NativeScript are placed inside the `Page` element.

Underneath the `Page` element, we've placed a `StackLayout` and `Label` element, which together display the text `Welcome to the Tekmo App` on the screen.

DEFINITION Stack Layouts are UI elements used to organize other UI elements within a page. The Stack Layout works alongside other UI elements that are nested inside.

DEFINITION Labels are a UI element used to display text visually within a page. Just like other UI elements.

The details of how stack layouts work (and even what they truly are) isn't important right now, so we're going to side-step the details. We'll cover the details in chapter 5, but for now, imagine that the stack layout makes all elements inside of it appear on the screen, and every page we create in this chapter needs a stack layout immediately underneath the `Page` element. Let's take a closer look at the `Label` element.

Labels are self-explanatory: they display text on a screen. There's not much more to say, except you use the `text` property of the `Label` element to set the text that you want to display.

Now that you've learned the basics about everything on the home page, open the updated app in your emulator and you'll see something like figure 4.6.

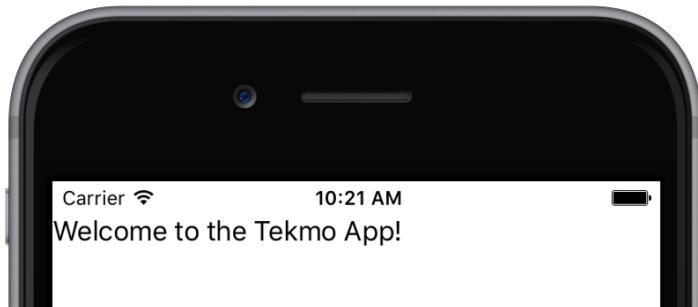


Figure 4.6 A label rendered in the Tekmo app on the iOS simulator.

How to run the tekmo app using the NativeScript CLI

In chapter 2, we introduced you to the `tns run` command. You can use the following CLI command to run the Tekmo app in the Android emulator or iOS simulator:

```
tns run android --emulator
```

```
tns run ios --emulator
```

For a detailed list of CLI commands, see appendix B.

Looking good, on to the rest of the Tekmo app!

4.2 Creating another app page

With only one page, the Tekmo app doesn't serve much of a purpose yet so it's time to add more content to the Tekmo app by creating another page.

4.2.1 Creating the About page

The second page we're creating is the About page. Add another file to the `views/about` folder named `about.xml` (figure 4.7).

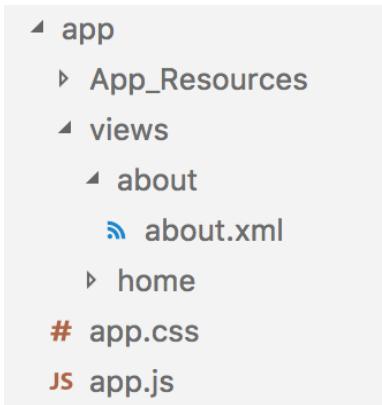


Figure 4.7 The views/about folder of the Tekmo after adding the *About* page.

TIP Remember, it's important to be consistent and organized, so we're using the conventions suggested in chapter 3, and adding a separate folder for each page of our app.

Just like when we created the Home page, we'll need to update the XML of the About page to give it content. Listing 4.5 contains the About page with several labels underneath the page and stack layout elements.

Listing 4.5 The views/about/about.xml file

```
<Page>
  <StackLayout>
    <Label text="Small company that wants to bring you the best in retro gaming!" />
    <Label text="Come visit us in Louisville, KY" />
  </StackLayout>
</Page>
```

Let's look at the page in your emulator. Even though we haven't learned how to navigate from the to the About page yet, let's use a quick trick to see the About page. Change the main module from views/home/home to views/about/about in the *app.js* file and run your app.

TIP If you're working on a specific page that would normally require you to navigate to it to test, set it as your default page. This helps to reduce your development time and instantly load the page in development. Just be sure to change it back when you're finished!

Figure 4.8 shows the About page after running your updated app in your emulator.

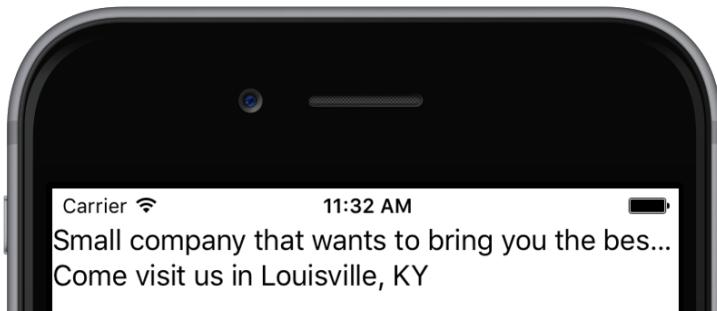


Figure 4.8 Multiple labels rendered in a Stack Layout element of the About page.

Right away, you have probably noticed something strange: the label text is being cut off. This can happen depending on the resolution and orientation of the device that the page is being rendered on, so it's something you'll continually need to think about when developing mobile apps.

NOTE Although NativeScript will let you write once and run on iOS or Android you will still need to be mindful of device resolutions. There are hundreds of devices in the marketplace and that means there are potentially hundreds of different resolutions that users are using your app in. Device resolution is something you will have to consider as you write your app. Don't worry though, I will show you some tricks later in this book to help make sure your app works correctly across a multitude of devices and resolutions!

Although you do need to consider text length and screen size, NativeScript makes it easy: just add the `textWrap` property to your labels. The following code shows you how to use the `textWrap` and ensure label text doesn't get cut off.

```
<Label textWrap="true" text="Small company that wants to bring you..." />
```

TIP Unless you're trying to restrict your text to a single line, you'll want to set the `textWrap` property of a label to true. Get in the practice of setting it immediately after creating your labels.

NOTE If you've noticed a difference between the text in our figures and text in code, you've got an acute eye for detail. We've purposefully truncated the text in code samples with ellipses to make it more readable, but figures will show the full text. We'll continue to do this throughout the book when it makes sense.

Setting the `textWrap` property to `true` will cause NativeScript to render the label element on multiple lines (the `textWrap` property of a label acts just like text wrapping does in your favorite text editor). Where the text wrapping of the label happens depends on the device that the page is being rendered on. In figure 4.9, you can see how the `textWrap` property behaves when rendering the About page on an iPhone 6 device.

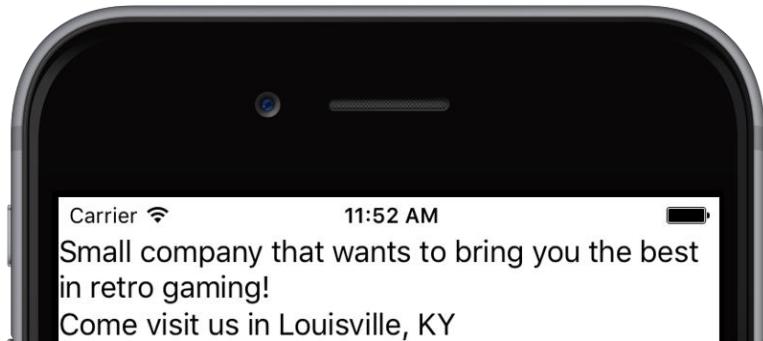


Figure 4.9 A label with the `textWrap` property set to true rendered on an iPhone 6 device

Wrapping text is one of the easy things you can do to account for multiple screen resolutions and orientations. For now, it is nice to know that NativeScript has your back again when bad things happen (ok text getting cutoff isn't the worst thing in the world but at least it is nice to know the NativeScript developers took this into account for you). As you continue to learn about NativeScript, we'll go into more detail on how to handle multiple platforms.

Now that you are finished creating the About page, it is time to talk about navigating between the Home and About pages.

NOTE Before continuing be sure to change your app's starting page back to `views/home/home` inside of the `app.js` file!

4.3 Navigating between app pages

Up until now, we've been living in the UI layer of the Tekmo app, creating XML files, and ignoring the business logic that lives in JavaScript. Now that we're ready to start navigating between pages, it's time to break out your JavaScript skills.

Like HTML applications, navigating between pages in a NativeScript app occurs in response to an action (typically, tapping on a link or button). In the next sections, you'll be learning how to add buttons to your pages, respond to the button being tapped by a user, and then navigating to another page. This may sound overwhelming at first, but it'll turn out to be very straight forward.

NOTE You may have noticed that we didn't say links and buttons are clicked in NativeScript apps. That's because clicking is what you do with a mouse, and most mobile devices don't have a mouse interface. Instead, you use your finger to tap the screen. Although these are two different terms for almost the same action, you might get an eye-roll from some hard-core mobile developers if you mistakenly say click instead of tap.

4.3.1 Adding a button to the Home page

Throughout the beginning of the book, you've seen me use buttons on pages. Previously I've said not to worry about the details of a button – thank you for waiting patiently! I hardly think a button needs an introduction, but let's be thorough and review what you should expect out of a button in NativeScript.

DEFINITION Buttons are a user interface element, having a visual and interactive aspect. Visually, buttons have text that is displayed on the screen. As a point of interaction, you can write business logic code in JavaScript that is run when a button is tapped. To create a Button element, use the XML code `<Button text="..." />`.

Let's carry forward the button concept to the Tekmo app, and place a button on the Home screen. The button should allow users to learn more about Tekmo by tapping it and then navigating to the About page. Listing 4.6 shows an updated version of the Home page for the Tekmo app, where we've added a button to the page.

Listing 4.6 Adding a button to the Home page

```
<Page>
  <StackLayout>
    <Label text="Welcome to the Tekmo App!" />
    <Button text="About" tap="onTap" /> // #A
  </StackLayout>
</Page>
#A The tap property tells NativeScript which JavaScript function to call when the button is tapped
```

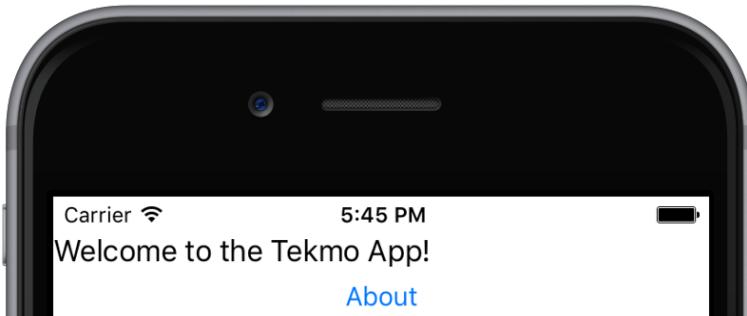


Figure 4.10 The Home page, complete with the About button.

As you can see in listing 4.6 and figure 4.10, like the label element, the button element has a `text` property that you can set to use to display what you want the button displays to the user.

NOTE When rendered on iOS the buttons look like a web link. If you are unfamiliar with iOS this is the current native styling of a button in iOS.

You may have also noticed the button element has an additional property: `tap="onTap"`. The `tap` property of the `Button` element is an *event* that is *raised* by the button; as you may have guessed the button raises an event named `tap` when the button is tapped. You can handle the `tap` event by setting the value of the `tap` property to the name of a function in your corresponding JavaScript file.

DEFINITION An event is an occurrence of something that happens with a NativeScript app. An event may occur because of user interaction with your app or because of a state that your app has transitioned to such as started, closing, closed, or out of memory. When an event occurs, it's referred to as being *raised*. When you configure your app to respond to a raised event, it's called handling the event. In NativeScript, you handle events through the JavaScript code you write in your app.

Ok, we just covered a lot there in a few sentences, and I promise we'll explain it in more detail in a minute. For now, let's focus on the button. Figure 4.11 helps you visualize the process of tapping a button and how the `onTap()` function is called.

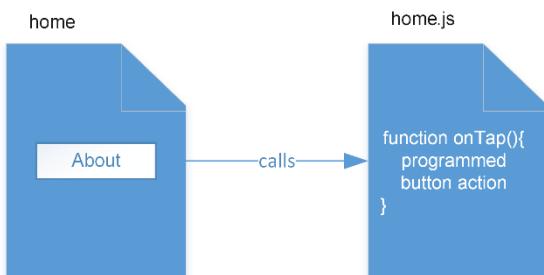


Figure 4.11 How tapping the About button calls a method defined inside of the corresponding JavaScript file.

Now that we've visualized and briefly explained how the `onTap()` function is called, let's get back to our concept of a NativeScript page: how did NativeScript know which file the `onTap()` function was in?

4.3.2 Button events

When the Home page is loaded in your app, do you remember what gets loaded? You may recall the answer from chapter 3 where you learned about page naming conventions within NativeScript. When a page is loading the NativeScript runtime looks for a correspond .js and .css file to load alongside the .xml file. In this case the runtime is looking to load `home.xml`, `home.js`, and `home.css`. Figure 4.12 shows how these three files are used when the home page is loaded.

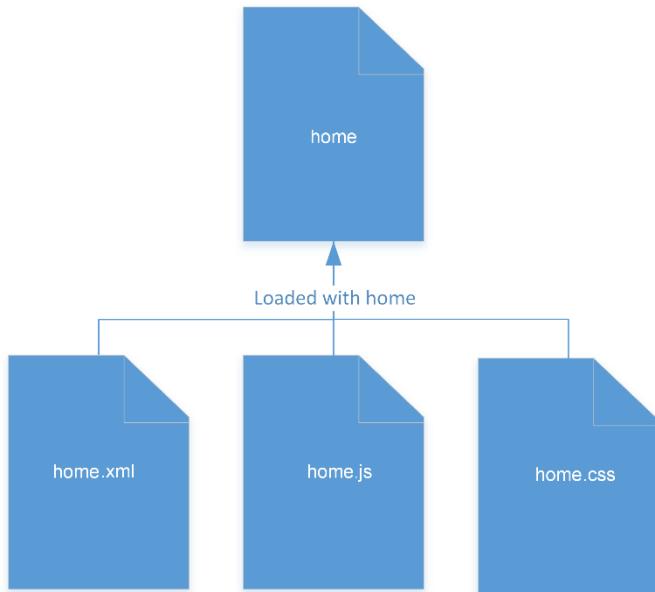


Figure 4.12 The three files that the NativeScript runtime loads when the Home page is loaded.

Back to our previous question. Assuming we have a button defined as `<Button text="About" tap="onTap" />`, how does NativeScript know which file the `onTap()` function is in? The answer is almost too easy, but for completeness, it looks in the corresponding JavaScript file for the page. In our case, that's the `home.js` file.

Now that we know how NativeScript finds the `onTap()` function, let's create the `onTap()` function in the `home.js` file.

TIP As I have previously discussed, it is important to maintain conventions when writing your app code. Implementing event handlers is another way you can use conventions to keep your code maintainable. We recommend using the appending “on” to the event name when handling events in your app. By adopting this convention, it will be much easier for you to maintain your app as it grows.

Listing 4.7 outlines the `onTap()` function definition and the navigation to the About page. There's a lot of new concepts in the code, so we'll be walking you through it below.

Listing 4.7 Implementation of the `onTap` function in the `home.js` file

```
var frameModule = require("ui/frame"); // #A

function onTap() {
    frameModule.topmost().navigate("views/about/about ");
}
```

```
exports.onTap = onTap; //#C
#A Get a reference to the NativeScript frame module which is used for navigation
#B Use the frame module to navigate to the About page
#C The function must be exported so the NativeScript runtime can access it from the UI
```

At the top of the home.js file, we load the frame module by requiring ui/frame. Navigation in NativeScript relies on the *frame* module.

DEFINITION The frame module is another module included in the NativeScript core modules, which is the collection of cross-platform abstractions used throughout NativeScript apps for UI and business logic code. The *frame* module is used to navigate between pages in your app by invoking the `navigate()` method on the topmost frame. If it helps, think of your app's frame as the outer wrapper that contains, loads, and navigates between pages.

DEFINITION The topmost frame is another cross-platform abstraction used to indicate the outer-most frame responsible for navigating to and loading new pages.

After the declaration for the *frame* module, you'll see the `onTap()` function definition in listing 4.7. Inside the function you'll see the *frame* module being used to get the topmost frame, and then navigating to the *About* page.

When the `navigate()` method is called, there's a little more going on behind the scenes in NativeScript, but let's keep it simple. In addition to the new page being loaded, NativeScript keeps track of a few things such as the list of previous pages, data (or variables) specifically passed to the new page, and directives on how to animate the transition to the new page once it's been loaded (figure 4.13).

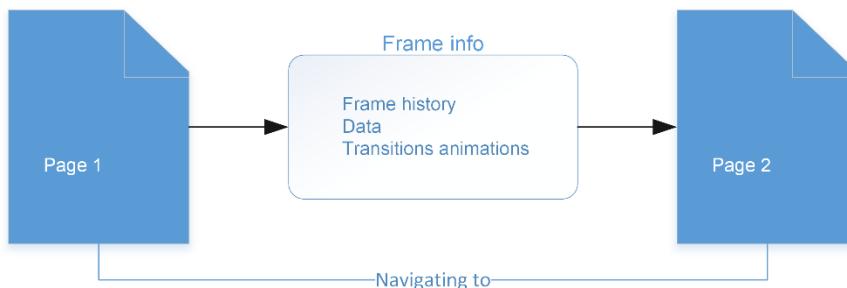


Figure 4.13 The frame module is used for navigation. When navigating with the frame module, frame information is transferred and stored within the page.

Now, that wasn't too difficult. With a few lines of code, you're able to respond to a button tap, load the *frame* module, and navigate to a new page. It really is that easy. Go ahead and run your app again and navigate to the *About* page by tapping the *About* button.

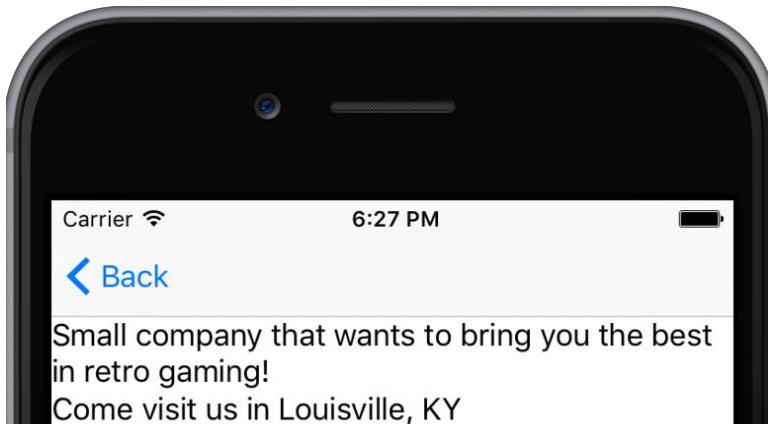


Figure 4.14 The About page after navigating to it with the frame module. The frame module uses native navigation on the Android and iOS platforms which allows the user to easily navigate back to the previous page.

After navigating to the About page, you may have noticed it looks a bit different now: a back link is automatically displayed at the top of the app. This is displayed because NativeScript is keeping track of your navigation history. In future chapters, we'll go into more detail on how you can use the frame module to navigate and further control the history and even send data between pages.

Before we dive into some of these more advanced scenarios, let's learn how to animate page navigation through something called a transition.

4.3.3 Applying transitions to page navigation

When you navigate between pages in a NativeScript app, the new page loads by sliding in from the right of the screen. If you ask us, this animation is a little plain. But, there's good news: you can easily change this animation by applying something called a page transition.

DEFINITION A page transition is an animation that occurs when you navigate from one page to another.

There are several different types of page transitions you can use in NativeScript, but you will want to be careful which transitions you use because only some are available for both Android and iOS. Table 4.1 show which transitions are available to each platform.

Table 4.1 Navigation transitions and the platforms they are available on

Transition	Platform Availability
curlUp	iOS
curlDown	iOS
explode	Android (Lollipop and later)

fade	Android, iOS
flipRight	Android, iOS
flipLeft	Android, iOS
slideLeft	Android, iOS
slideRight	Android, iOS
slideTop	Android, iOS
slideBottom	Android, iOS

Most of the transitions available to you are self-explanatory. In the Tekmo app you are going to use the `slideBottom` transition since it is available on both Android and iOS. As the name implies, the `slideBottom` transition makes the page that you are navigating away from appear to "slide" towards the bottom of the screen. Likewise, the `flipRight` transition makes the page that you are navigating away from appear to be "flipping" towards the right.

NOTE When you transition back to the previous page the opposite transition will be applied. In the upcoming example, you will implement the `slideBottom` transition; when you navigate back to the Home page, the `slideTop` transition will apply automatically.

Enough about what transitions look like; let's start using them. Update the `home.js` file to contain the code in listing 4.8. As you're updating the code, you'll notice a new object named `navigationEntry` is used and passed into the `navigate()` function instead of passing the name of a page.

DEFINITION The navigation entry object is an interface called `NavigationEntry`. This interface defines the data that is passes from page to page when the `navigate` method is called from the frame module. It allows you to define what page you want to navigate to, page transition animations, and how to handle the navigation history.

Listing 4.8 Applying the `slideBottom` transition when navigating to the `About` page from the `Home` page

```
var frames = require("ui/frame");

function onTap() {
    var navigationEntry = { //A
        moduleName: "views/about/about", //B
        transition: { //C
            name: "slideBottom" //C
        }
    };

    frames.topmost().navigate(navigationEntry);
}
```

```
exports.onTap = onTap;
#A The NavigationEntry variable represents an instance of the NavigationEntry interface defined by NativeScript; there are several optional properties of the object that you can specify
#B The moduleName is an optional property that specifies the page that you want to navigate to
#C The transition property will set the transition that you want to apply when navigating to the module you defined. The name property of the transition object is the transition that you want to use.
```

The `navigationEntry` object contains several properties used by the `navigate()` function to manage the navigation from page to page. In listing 4.8, we use the `moduleName` and `transition` properties, which reference the page to navigate to and the page transition to use during navigation.

TIP In chapter 3, you learned about platform-specific file-naming conventions. Navigation transitions are a good use case for a platform-specific JavaScript files because not all transitions are available on both Android and iOS.

Unfortunately for you, the world has not figured out how to put animations in books yet so you can't really visualize the `slideBottom` transition (I guess you will just have to try it yourself).

4.4 Summary

In this chapter, you learned how to:

- Implement a page in NativeScript
- Handle the tap event of a button and implement corresponding business logic when the button is tapped
- Navigate between pages in a NativeScript app using the frame module
- Bind and respond to application-wide events such as the application-launch event

4.5 Exercise

1. Modify the Tekmo app and apply the fade animation when you navigate to the About page
2. Add a button to the About page that will navigate to the Home page

4.6 Solutions

1. Update the `home.js` file:

```
var frames = require("ui/frame");
var navigationEntry = {
    moduleName: "views/about/about",
    transition: {
        name: "fade"
    }
};

function onTap() {
    frames.topmost().navigate(navigationEntry);
```

```
}
```

```
exports.onTap = onTap;
```

2. Update the about.xml and about.js file:

Add to about.xml:

```
<Button text="Home" tap="onTap"></Button>
```

Add to about.js:

```
var frames = require("ui/frame");

function onTap() {
    frames.topmost().navigate("views/home/home");
}

exports.onTap = onTap;
```

5-Jan-2019

5-Jan-2019

5

Understanding the basics of app layouts

This chapter covers

- Why it's important for an app's design to match its function
- How to organize and layout the user interface (UI) of a NativeScript app with a Stack Layout
- How to use common UI elements (labels, buttons, and text boxes)
- How to create scrollable pages using the Scroll View

In Chapter 4, you learned how to navigate between the pages of a NativeScript app using XML and JavaScript code. You also began to create a mobile app for the Tekmo company. In this chapter, you'll continue to refine Tekmo's app while learning about UI layouts.

5.1 Understanding NativeScript layouts

Just creating pages for your app and navigating between them isn't enough. Mobile app development is also about creating compelling user experiences to accomplish a purpose. This means you need to present and organize your UI in different ways depending upon what the users of your app do.

For example, let's assume you design a fitness app with a dashboard showing you how much you exercise each day, similar to the fitbit app shown in figure 5.1.

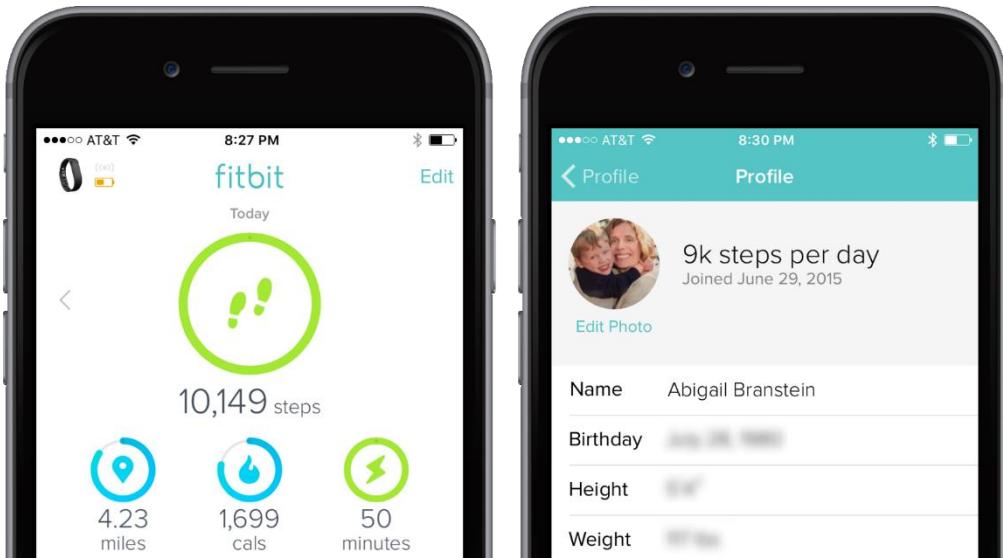


Figure 5.1 Two different UI layouts, with charts data points displayed on the left dashboard page, and more tabular data entry on the profile page, as seen in the fitbit app on iOS

On the dashboard are several different charts and graphs displaying your daily progress. The charts and graphs are oriented in different patterns on the screen to catch the eye of a user, drawing attention to the most important data first (your step count). On a different page, users complete a personal profile, including their name, birthdate, height, weight, and fitness goals. Because of the difference in information displayed on the profile page, the UI elements are arranged in a vertical stack.

Right now, designing a mobile app with these various organizational layouts may seem daunting, but don't worry. In this chapter and subsequent chapters, you'll be learning how to organize UI elements. In fact, NativeScript makes organizing your UI easy with a collection of UI elements called *layouts*.

DEFINITION A layout is a special UI element that instructs and informs your app how to organize other UI elements on a mobile device's screen. If you want a button to *dock* (or always be displayed) at the bottom of your screen, or if you'd like to organize several buttons in a grid-like collection of rows and columns, there are layouts to help you.

Because NativeScript layouts are UI elements, they are defined in XML. Although I call layouts a UI element, they are not displayed on a mobile device's screen when a page is loaded. Instead, they affect the location and arrangement of UI elements placed within the layout. Because layouts will *contain* other UI elements, they are referred to as *layout containers*.

In listing 5.1, I have included a *Stack Layout* with several nested UI elements. You'll learn about *Stack Layouts* in just a moment, so you might not understand how a *Stack Layout* works right now, but take

note that when UI elements are placed inside of a layout container, they are arranged and displayed on the mobile device screen in various ways (according to the rules of their container).

Listing 5.1 Stack Layout container with several UI elements

```
<StackLayout>
    <Button text="Everything is awesome..." />          #A
    <Button text="when you're using NativeScript!" />      #B
</StackLayout>
#A Stack Layout is a UI element, but will not be displayed on the screen
#B UI elements contained within the Stack Layout element will be displayed, per the rules imposed upon them by the Stack Layout
```

In total, there are five different layouts: stack-layout, grid-layout, wrap-layout, dock-layout, and absolute-layout. I'm not going to go into detail about all the layouts, but instead focus on the most common layouts you'll use when building your first several apps: stack and grid. You'll learn about the stack layout in this chapter, and the grid layout in chapter 6.

NOTE Every NativeScript app page you create will have a layout, but you're not limited to having a single layout on a page. In later chapters, I'll show you how to build more complex UIs by *nesting* (or combining) two or more layouts together.

TIP The NativeScript documentation on layout containers is an excellent resource: <https://docs.nativescript.org/ui/layout-containers>. The explanation of all five layouts is a great quick-reference guide. Bookmark it now.

5.1.1 Layouts and screen pixels

As you're learning about layouts, it's important for you to understand some basics of how NativeScript (and most mobile devices) display UI elements on the screen. In chapter 3, you learned about *screen resolution* and *DPI*.

DEFINITION Screen resolution is a measure of the number of pixels on a screen, usually described in the form of width x height. For example, a screen resolution of 640x480 means the screen is 640 pixels wide and 480 pixels high.

DEFINITION Dots per inch (DPI) is a measure of dot (or pixel) density on a mobile device's screen. In mobile apps, it's used to describe the number of pixels appearing in an inch of screen along the width or height. Most devices have the same DPI in both horizontal and vertical directions, so the DPI is described as a single number. For example, 400 DPI would mean 400 pixels per inch exist in each row and column of pixels.

Although mobile devices have varying screen resolutions and DPI, all devices have rectangular screens organized in rows and columns of pixels. Each pixel on the screen can be referenced by using a coordinate of (X, Y), with X representing the number of pixels from the left of the screen and Y representing the number of pixels from the top of the screen.

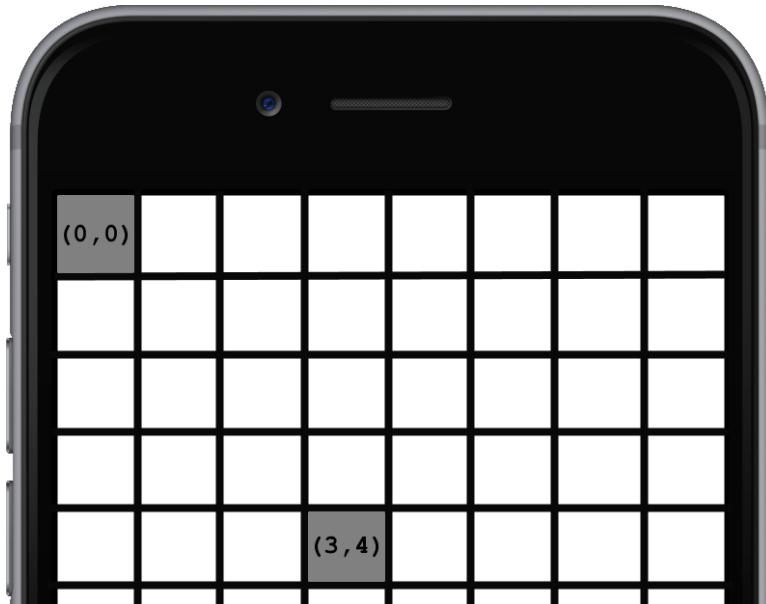


Figure 5.2 The leftmost top pixel is referenced by the (0, 0) coordinate. The fourth pixel from the left and fifth from the top is referenced by the (3, 4) coordinate.

Figure 5.2 shows how the leftmost top pixel can be referenced by the coordinate of (0, 0). Similarly, the pixel in the fourth column and fifth row down can be referred to using the coordinate (3, 4).

NOTE You'll notice that the first row and column on the screen is referred to using the number zero. You're probably familiar with this practice of counting from zero instead of one, but I just like to call it to your attention.

When you use NativeScript layouts to organize your UI elements, you won't spend a lot of time thinking about individual pixels (at first). However, as you learn about each layout using screen coordinate to reference pixel is an easy way to explain how NativeScript places UI controls on the screen. You'll also use the concepts you learned about screen coordinates later in this book when you start styling apps with CSS.

5.2 Stack Layout

Earlier in this chapter, you learned that UI elements located inside of layout containers are arranged on the screen based upon the rules governing the associated container. Listing 5.2 and figure 5.3 shows the same Stack Layout you saw previously in code and on a mobile device.

Listing 5.2 Stack Layout container with several nested UI controls

```
<StackLayout>
  <Button text="Everything is awesome..." />
  <Button text="when you're using NativeScript!" />
</StackLayout>
```

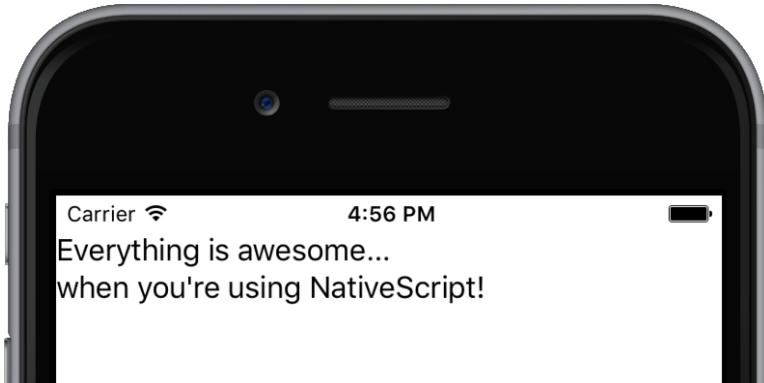


Figure 5.3 A Stack Layout displays its nested UI elements "stacked" on top of each other.

Staying true to its name, the *Stack Layout* organizes its nested UI elements by "stacking" each child element on top of each other. When the two *Buttons* are placed inside of the *Stack Layout*, they are placed on the screen in the same order they appear in XML. NativeScript does a lot behind the scenes to make the *Buttons* display in a stack, and the details aren't important. However, at a high level, each UI elements contained within a *Stack Layout* are placed on the screen, sequentially. The first *Button* with the text of "Everything is awesome..." is placed on at the top of the screen, then the second *Button* is placed beneath.

TIP Stack Layouts are the most common layout container used in NativeScript apps, and I find myself using them everywhere. On most pages, the very first UI control I add is a Stack Layout. As you continue to learn about other layout containers and wonder which layout you should use, start with a Stack Layout.

5.2.1 Adding content to the Tekmo app using Stack Layouts

Now that you've learned how to use the *Stack Layout*, let's continue to build out the Tekmo app. In chapter 4, I created the Tekmo app and added two pages: Home and About. I also added basic navigation from the Home page to the About page. In this chapter, I'll be adding a third page, Contact Us, and also fleshing out the About Us page with additional UI elements. Figure 5.4 shows what I'll be building up to through the rest of this chapter. You can follow along, incrementally building each page with me or you can download the full code listing from Github at <https://github.com/mikebranstein/TheNativeScriptBook>.

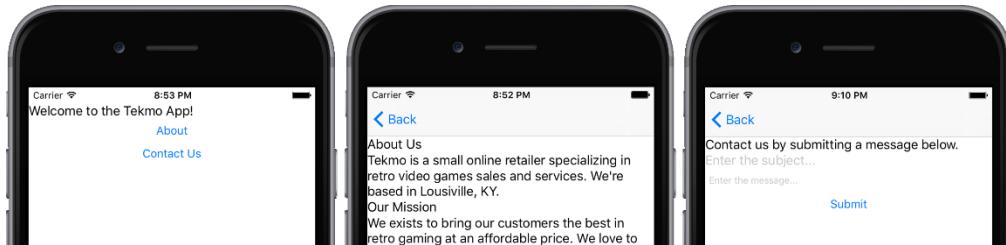


Figure 5.4 Final chapter 5 version of the Tekmo app incorporating layouts and additional UI elements for the Home, About, and Contact Us pages.

ADDING CONTENT TO THE ABOUT US PAGE

In chapter 4, I added a brief summary of the Tekmo company to the About Us page, as seen in listing 5.3 and figure 5.5.

Listing 5.3 The About Us page created in chapter 4 with a brief description of the Tekmo company

```
<Page>
<StackLayout>
    <Label text=="Small company that wants to bring you...">
        textWrap="true" />
    <Label text=="Come visit us in Louisville, KY" />
</StackLayout>
</Page>
```

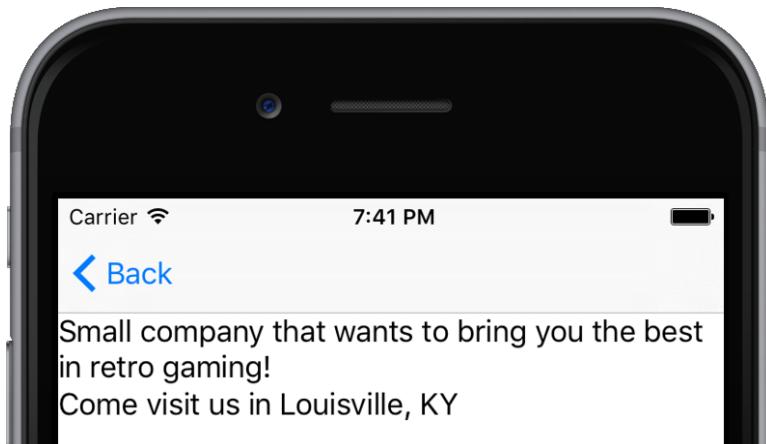


Figure 5.5 The About Us page from chapter 4 before I've added addition content to the Stack Layout.

In HTML applications, About Us pages typically have a heading, a mission statement, and a bit longer text telling visitors about the company's history. I'm going to add some of these elements to the Tekmo

app by placing a several more Labels within the existing Stack Layout. Listing 5.4 shows the additional Label elements added. Figure 5.6 then shows the full layout of the About Us page.

Listing 5.4 A heading, mission statement, and company history added to the About Us page

```
<StackLayout>
    <Label textWrap="true" text="About Us" />
    <Label textWrap="true" text="Tekmo is a small online retailer..." /> #A
    <Label textWrap="true" text="Our Mission" /> #A
    <Label textWrap="true" text="We exist to "" /> #A
    ...
</StackLayout>
#A Text has been truncated, but shown in figure 5.5
```

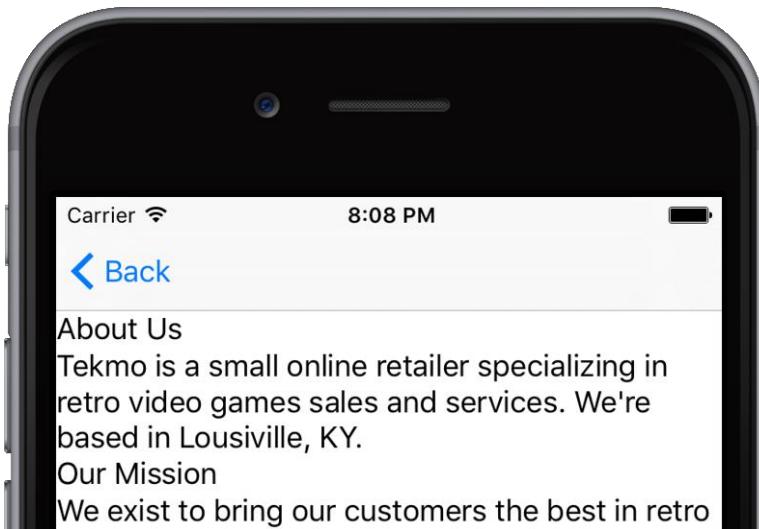


Figure 5.6 The About Us page showing general information about Tekmo and the Tekmo mission.

Now that I've added more text to the About Us page, the Tekmo app is really starting to come together. But wait. There's something subtle that is easily overlooked. Looking at the bottom of the About Us page (figure 5.7), you'll notice that some of the text is now scrolling off the bottom of the screen.

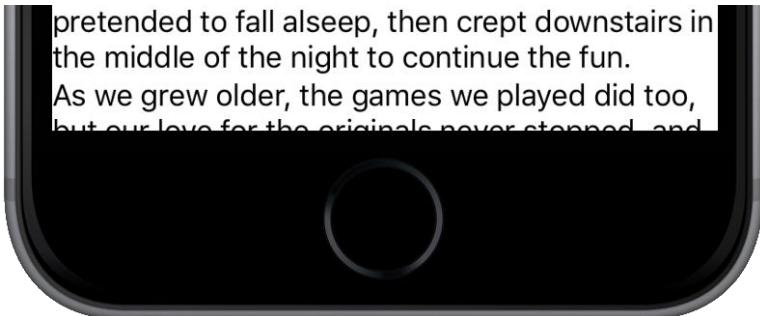


Figure 5.7 The text on the About Us page scrolls off the bottom of the screen.

In chapter 4, you learned how to account for text scrolling off the page horizontally by using the `textWrap="true"` attribute of the `Label` element, but now text is scrolling off the page vertically. Let's learn how to fix this problem.

5.2.2 Scrolling pages

On mobile devices, when screen contents scroll off the bottom of the screen, your natural inclination is to swipe the screen and move the content. But wait: if you try to swipe the screen on the Tekmo app's About Us page, nothing will happen. By default, NativeScript pages do not allow you to swipe and scroll content. To enable scrolling, you'll need to use the *ScrollView* UI element.

TIP To allow page content to be scrollable, add a *ScrollView* to a page.

ScrollView are like layouts because they aren't displayed on a page and they are container objects. Listing 5.5 and figure 5.8 show the *ScrollView* in action on the About Us page of the Tekmo app.

Listing 5.5 About Us page with a ScrollView, to allow its contents to scroll by swiping the screen

```
<Page>
  <ScrollView>      #A
    <StackLayout>    #B
      ...
    </StackLayout>   #B
  </ScrollView>
</StackLayout>
#A The ScrollView is the first UI element on the page
#B Scrollable UI code goes here, starting with our layout container and the contents
```

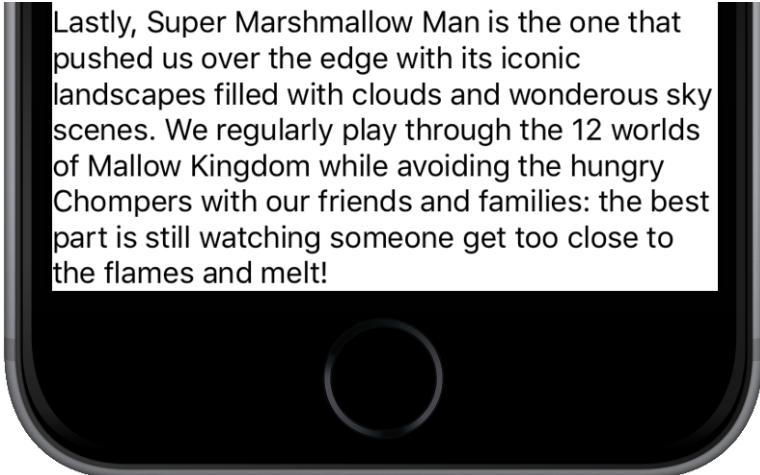


Figure 5.8 Adding a *ScrollView* to the Home page allows users to scroll to the bottom of the page's content by swiping.

To make a page's content scrollable, the *ScrollView* elements should be the first element added to a page. Inside of the *ScrollView*, you should place the *StackLayout* and its nested elements. As shown in figure 5.8, once the *ScrollView* is added, you can use your finger to swipe and scroll page content.

TIP Pay attention to user experience when adding scrollable content to an app! Don't scroll a subset of your page on smaller screen devices. Don't scroll everything (unless you should) on larger tablets. If you want to read more about UI and app experience design, check out the iOS Human Interface Guidelines at <https://developer.apple.com/ios/human-interface-guidelines> or Android's Design Guide at <https://developer.android.com/design>.

When you add a *ScrollView* to a page, you do not have to place the *ScrollView* at the very top of the page. In fact, it's possible to place a *ScrollView* around a portion of your page's content. For example, imagine you have a page designed like a news article, as seen in figure 5.8. The page has a large heading at top taking up about 50% of the total screen height. The remaining 50% of the screen contains the article text.



Figure 5.9 A news article-like page with a large headline taking up 50% of the vertical space.

On this page, you could place a *ScrollView* around the entire page or around the article text. But which choice is right? There may not be a *right* choice, but I recommend making the entire page scrollable. Why? My answer comes from more feeling than logic. Mobile apps are about good user experience, and I believe that if I were to only make the article text scrollable, it would make my finger swipe gestures feel restricted, especially on smaller phones. When scrolling, I typically like to make broad sweeping strokes, or moderately long, quick flicks with my finger. If the scrollable space is too small, broader strokes won't feel natural. So, on smaller screens, it makes sense to scroll the entire page.

But what about a larger tablet screen? On a tablet, the article headline takes up less vertical space because of the larger screen size, leaving more room for the article text. Because there is more room for the article text, I would scroll only the text, leaving the headline on the page at all times.

Keep in mind, there's no *right* or *wrong* answer about where and how to use the *ScrollView*. However, it's not uncommon to change how you use a *ScrollView* on a page once you've tested your app on a real device.

TIP Test your app on a physical device. That's the only way you'll be able to tell if your design choices work.

TIP Don't be the only person to test your app on a physical device. As the app's developer, you'll miss subtleties that your friends or colleagues will notice immediately. Install your app on a friend's phone or pass your phone around the office to elicit feedback.

Now that you've learned how to add text to a page, wrap the text to multiple lines, and ensure your page's content is scrollable, you may think the page is plain. Well, sorry to break it to you, it is. But, don't worry. You'll learn how to style the page differently in a later chapter.

Just in case you haven't been able to follow along, listing 5.6 contains the complete code for the About Us page (with long lines truncated).

Listing 5.6 Complete About Us page code for the Tekmo app

```
<Page>
  <ScrollView>
    <StackLayout>
      <Label text="About Us" />
      <Label textWrap="true" text="Tekmo is a small online..." />
      <Label text="Our Mission" />
      <Label textWrap="true" text="We exist to bring..." />
      <Label text="History" />
      <Label textWrap="true" text="In the early 90's, it all..." />
      <Label textWrap="true" text="After Rescue Pups, it was..." />
      <Label textWrap="true" text="As we grew older, the games..." />
      <Label textWrap="true" text="After many year, we all started..." />
      <Label textWrap="true" text="We remembered how cool it was to..." />
      <Label textWrap="true" text="And then there was Vampire..." />
      <Label textWrap="true" text="Lastly, Super Marshmallow Man..." />
    </StackLayout>
  </ScrollView>
</Page>
```

5.2.3 Using textboxes and providing feedback to users

The next page of the Tekmo app I'll work on is the Contact Us page. In HTML applications, Contact Us pages allow visitors to send a message or request additional information. In this section, you'll learn how to use textboxes and buttons to create a UI for users to contact Tekmo via the app.

Let's begin and add the Contact Us page to the Tekmo app by creating a folder beneath the app's `views` folder named `contact-us`. Figure 5.10 shows this folder with the corresponding XML and JavaScript page files.

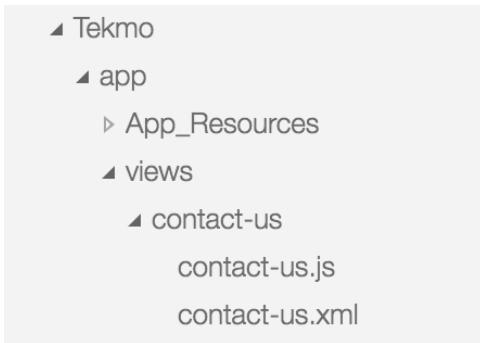


Figure 5.10 The app structure of the Tekmo app, after adding a Contact Us page to the views folder.

Before I start to add textboxes and buttons to the Contact Us page, let's review the XML code in listing 5.7. Now that you've learned about the *ScrollView*, *Stack Layout*, and *Label* elements, I've added them to the page.

Listing 5.7 Contact Us page with a ScrollView added

```
<Page>
  <ScrollView>
    <StackLayout>
      <Label textWrap="true" text="Contact us by submitting a
        message below." />
    <StackLayout>
  </ScrollView>
</Page>
```

On the Contact Us page, I want to allow users to enter a message subject and message body. Once they've entered this information, they should be able to submit the message to Tekmo by pressing a button.

ADDING TEXTBOXES TO THE CONTACT US PAGE

Just as in HTML applications, NativeScript apps have textbox UI elements called *Text Fields* and *Text Views*.

DEFINITION A *Text Field* is a single-line textbox, like an HTML `input` element with `type="text"`. To add a *Text Field* to a page, you use the `<TextField />` XML element.

DEFINITION A *Text View* is a multi-line textbox, like an HTML `textarea` element. To add a *Text View* to a page, you use the `<Text View />` XML element.

As you're planning your UI, it's important to know ahead of time if the textbox you'd like to show should be a single line or multiple lines. The Contact Us page will use both a *Text Field* and a *Text View*, as shown in listing 5.8 and figure 5.11.

Listing 5.8 Contact Us page with a Text Field and Text View added

```
<StackLayout>
    <Label textWrap="true" text="Contact us by submitting a message below." />
    <Label text="Subject" /> #A
    <TextField /> #B
    <Label text="Body" /> #A
    <TextView /> #C
</StackLayout>
#A You should add descriptive text before each textbox so users know what to place in each textbox
#B The single-line textbox for the message subject
#C The multi-line textbox for the message body
```

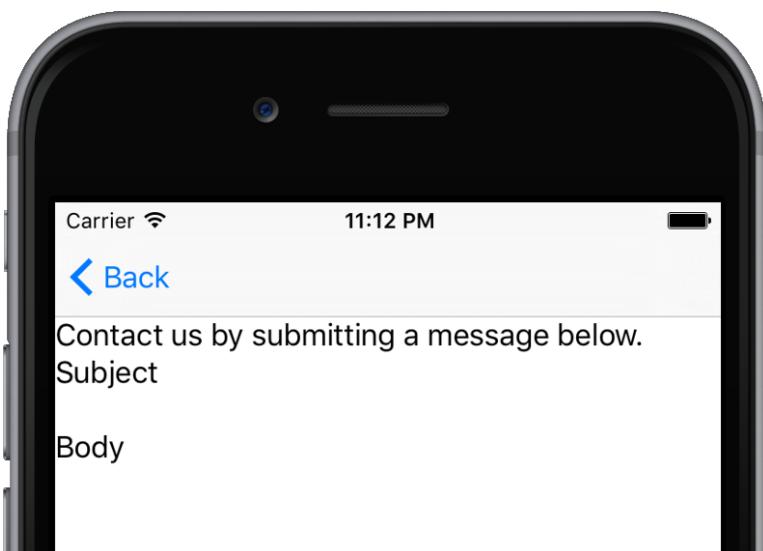


Figure 5.11 The Contact Us page with a message subject and body using a single-line *Text Field* for the subject and a multi-line *Text View* for the body.

When you add *Text Fields* and *Text Views* to your app, you should tell your users the purpose of each. One way of doing this is to place a label near the textbox. In listing 5.8, I placed a Label above each textbox to let my users know the first textbox should be used to enter the subject of the message, and the second textbox should be used to enter the body of the message.

Although placing Labels near a textbox can be an effective way of communicating the intent of the textbox, this approach can be ineffective (especially on iOS). On the iOS platform, the textboxes do not have any visual cues to tell the user a text box exists. Take figure 5.12 as an example: can you tell there is a textbox below the *Subject* and *Body Labels*? I can't either!

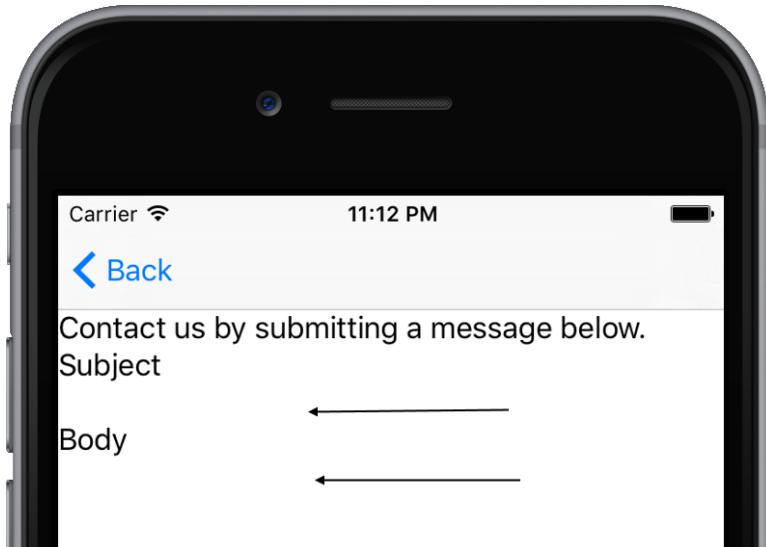


Figure 5.12 The Contact Us page with arrows pointing out the textboxes for Subject and Body. It's impossible to see because the textboxes have no visual cues letting you know they're there.

Because it may be difficult to see textboxes on the screen, I am very purposeful in how I identify textboxes. My preferred method of describing the purpose of a textbox is to use a *hint*, as shown in figure 5.13.

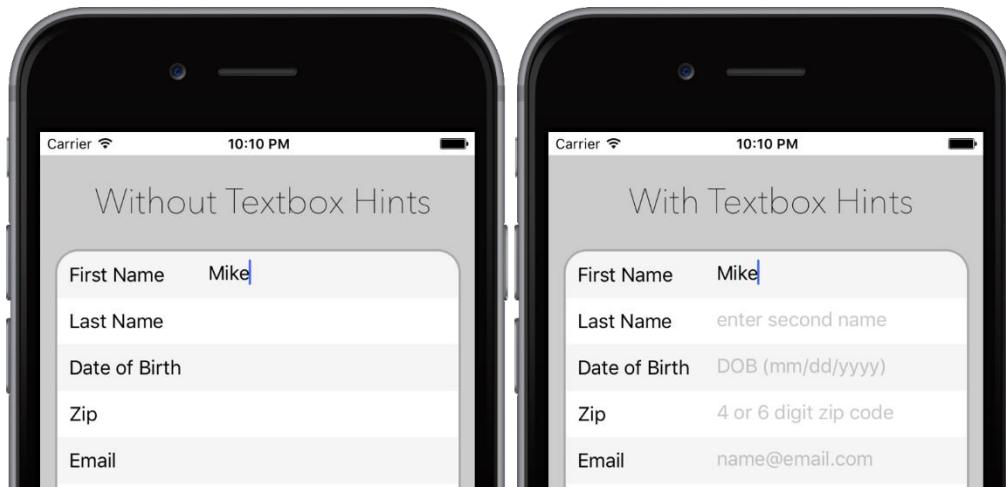


Figure 5.13 Comparison of two pages with and without hints. The page on the left does not have hints, and the page on the right does. Hints provide context to users on what type (or format) of data you'd like entered into a textbox.

DEFINITION A hint is a way of telling NativeScript to temporarily place text inside a textbox, as long as a user hasn't typed text into the textbox. For example, if a textbox hint is "Enter a subject," the text "Enter a subject" will appear in the empty textbox. When you tap textbox to enter text, the temporary text "Enter a subject" disappears, allowing you to type. Entering text into the textbox will remove the temporary text. Deleting all of the text you entered into the textbox will once again display the temporary text, "Enter a subject." To set a textbox hint, add a hint attribute to a *TextField* or *TextView*: <TextField hint="Enter a subject" /> or <TextView hint="Enter a message" />.

TIP Hints are a subtle way of making an app more useable. Two common uses for hints are describing what should be entered into a textbox (for example, "Enter a subject") and describing the expected formatting of the text entered into the textbox (for example, "name@domain.com" for an email address).

TIP If you use a hint effectively, you can often remove descriptive labels from your UI. By doing so, an app may look less cluttered and more visually appealing.

Hints can be powerful UI additions that not only make your pages easier to use, but also more visually appealing. Yes, "visually appealing" is completely subjective, but I'd rather use an app that dictates and guides me through using it properly and efficiently than guess where and how to enter data.

Because it's a good idea, I'm going to add hints to the Contact Us page of the Tekmo app. Listing 5.9 and figure 5.14 show how I use hints to give app users information on what should be entered into the subject and body textboxes. I also removed the descriptive Label elements above each textbox, because the hints provide enough content to the user.

Listing 5.9 Contact Us page using hints instead of descriptive labels

```
<StackLayout>
    <Label textWrap="true" text="Contact us by submitting a
        message below." />
    <TextField hint="Enter the subject..." /> #A
    <TextView hint="Enter the message..." /> #A
</StackLayout>
#A Hints can provide a way of giving users information on the expected purpose of textboxes, while also allowing
you to remove descriptive labels
```

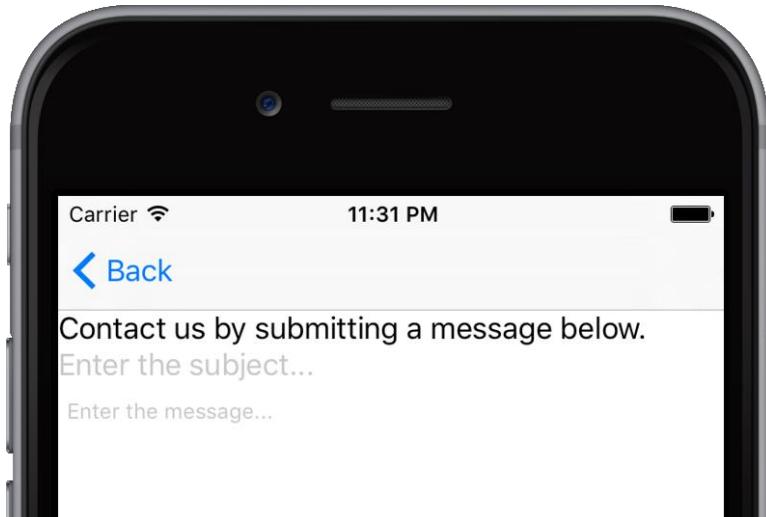


Figure 5.14 The Contact Us page with descriptive labels removed. The hints provide enough information for the user to understand the purpose of each textbox without the labels.

Although you've learned about the basics of a *Text Fields* and *Text Views*, there are many more properties like the *hint* property that are at your disposal as a NativeScript developer. I can't cover them all in this chapter, but you will learn more about these textboxes throughout the book. If you want to learn more now, read through the detailed API documentation at <https://nativescript.org>.

ADDING A SUBMIT BUTTON TO THE CONTACT US PAGE AND PROVIDING USER FEEDBACK

After a user enters a message subject and body into the textboxes on the Contact Us page, it's customary to provide a way for the users to actually send the information to you. Just as most HTML applications allow you to submit information by pressing a button, I want users of the Tekmo app to submit their message by pressing a submit *Button* within the app. In listing 5.10 and figure 5.15, I've added a *Button* to the *Stack Layout*.

Listing 5.10 Adding a Button to the Contact Us page to submit the message subject and body

```
<StackLayout>
  <Label textWrap="true" text="Contact us by submitting a
    message below." />
  <TextField hint="Enter the subject..." />
  <TextView hint="Enter the message..." />
  <Button text="Submit" tap="onTap" /> #A
</StackLayout>
```

#A When you tap the Submit Button, a function named "onTap" will be invoked by NativeScript.

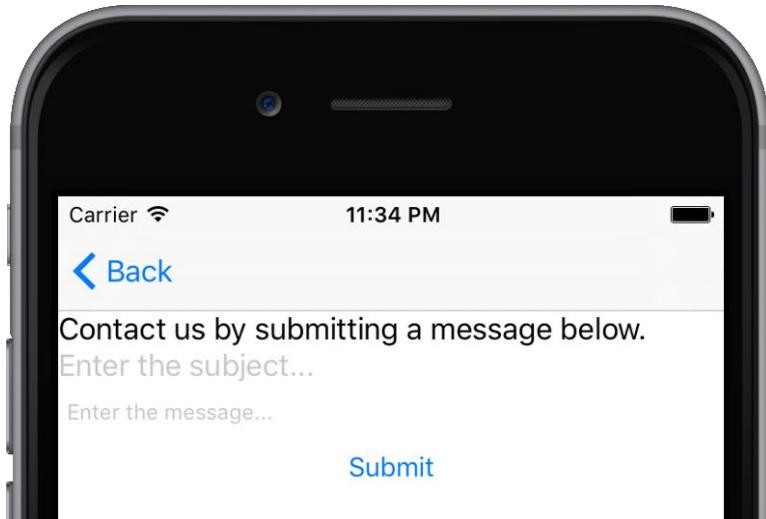


Figure 5.15 The Contact Us page, complete with a Button to submit the message to Tekmo.

You've already learned about *Buttons* and a *Button's tap* event in chapter 4, so you should recognize the *tap* event handler definition in listing 5.11. What you may not be familiar with yet is the body of the `onTap()` function. Ideally, I would send the entered information to Tekmo via email or by sending a message to a server, but I'm going to purposefully skip the "send" part. I'll revisit sending data and messages in a future chapter, so you won't miss out. But for now, I'll assume the message was sent, and notify the user it was sent successfully. Let's take a closer look at it together in listing 5.11.

Listing 5.11 Providing feedback to the user when the Submit Button is tapped

```
var dialogsModule = require("ui/dialogs"); #A

function onTap(args) {
    console.log("submit button tapped"); #B

    dialogsModule.alert("Your message has been sent."); #C
}
exports.onTap = onTap;

#A To provide the user with feedback that the button has been tapped, load the "ui/dialogs" module, which
contains an alert dialog to display a message to the user.
#B In your first several apps, it's a good idea to leave yourself a trail of bread crumbs to debug a misbehaving
app. Logging a message to the console is an easy way to do this.
#C Provide visual feedback to the user by displaying an alert dialog with a customized message. If we had
incorporated code to send an email or message to a server, it would appear directly prior to this alert.
```

TIP You may have noticed the `onTap()` event handler function name isn't very descriptive. We recommend using a naming convention like `onSubmitTap()` so your code is more readable.

Although you've seen the tap event handler before, I've introduced several new concepts. There is a lot going on, but don't panic. Let's unwrap it by starting at the first line of the `onTap` function, then we'll go back to the top and learn about the new module I introduced.

The first line of the `onTap` function references a variable named `console`. At first glance, you may be confused because nowhere in my code did I declare the `console` variable. Well, you're right, I didn't declare the `console` variable. However, that's OK because `console` is a global NativeScript variable. Let me explain.

When you're developing a NativeScript app (or any app), it's usually a good idea to instrument your app by leaving behind diagnostic messages (or bread crumbs). By doing so, you can re-trace your steps when something goes wrong. NativeScript provides a specialized module named `Console`, that provides you with ways to output diagnostic data and message to the Android emulator and iOS simulator.

NOTE When you're running your app in the Android emulator and iOS simulator, messages and data logged with the `console` module are displayed to the terminal or command prompt.

As you learned in chapter 4, to use a NativeScript module, you need to import it using the `require()` syntax. However, because you will use the `console module` on almost every page, NativeScript automatically loads it on your behalf into a global variable named `console`.

TIP Use the `Console` module extensively to instrument your app by outputting messages and key events. Even if you don't think you'll need the information, I guarantee you'll thank yourself in the future when you're trying to debug your app.

Now that you understand the purpose of the `Console` module, the first line of the `onTap` function is clearer. I'm using the `Console` module to log a diagnostic message to the emulator/simulator.

Let's look back at the other new module I introduced on the first line of listing 5.11: `var dialogsModule = require("ui/dialogs")`. This command imports the `Dialogs` module.

DEFINITION The `Dialog` module gives you a collection of `Dialogs` that provide visual feedback to users, typically in response to them performing an action. Although there are many different types of `Dialogs` that look and act in different ways, all `Dialogs` have three things in common when displayed on the screen: a heading (or title), a message, and one (or more) buttons prompting the user to acknowledge the message, close the `Dialog`, or perform another action.

DEFINITION An `alert` `Dialog` is a type of `Dialog` that displays a simple message and single button. Pressing the button closes the `alert`. `Alert` `Dialogs` are akin to a `JavaScript alert()` message in HTML applications.

With a better understanding of the `Dialog` module, you can understand how `dialogsModule.alert("...")`; displays a `Dialog` alerting the user that their message has been sent to Tekmo. Figure 5.16 shows how an `alert` `Dialog` is displayed to the user, providing visual feedback when the `Submit Button` is tapped.

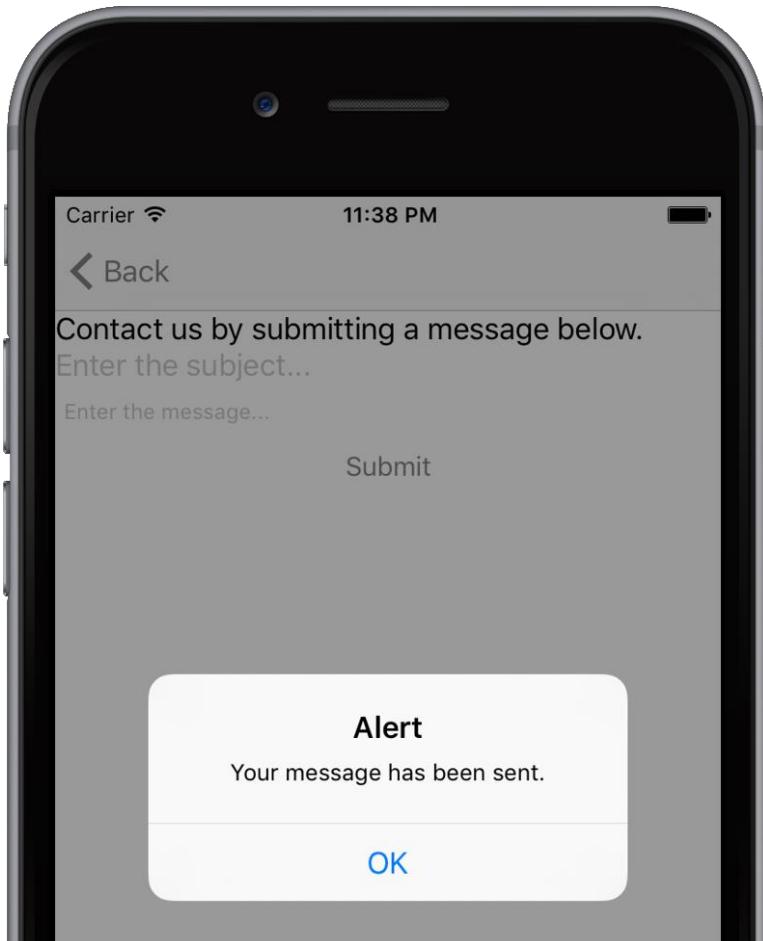


Figure 5.16 A dialog is displayed to the user after the Submit Button is tapped, alerting them that their message has been sent.

TIP Providing users with feedback when they interact with an app is an important aspect of mobile app development. There are many ways of providing feedback to a user within a mobile app: sounds, vibration, and visually.

At times, certain forms of user feedback will work better than others. In this book, I won't go into detail on which form of user feedback is best (and why). However, I do feel compelled to teach you that providing feedback to users is incredibly important. Later in the book, you'll learn about other facets of the *Dialog* module, and several other mechanisms for providing feedback to users. Stay tuned!

COMPLETE CODE LISTING FOR THE CONTACT US PAGE

I've finished adding UI elements and JavaScript code to the Contact Us page of the Tekmo app. In case you haven't been able to follow along, listings 5.12 and 5.13 include the complete XML and JavaScript code for the Contact Us page.

Listing 5.12 Complete XML code listing for the Contact Us page

```
<Page>
  <StackLayout>
    <Label textWrap="true" text="Contact us by submitting a
      message below." />
    <TextField hint="Enter the subject..." />
    <Text View hint="Enter the message..." />
    <Button text="Submit" tap="onTap" />
  </StackLayout>
</Page>
```

Listing 5.13 Complete JavaScript code listing for the Contact Us page

```
var dialogsModule = require("ui/dialogs");

function onTap(args) {
  console.log("submit button tapped");

  dialogsModule.alert("Your message has been sent.");
}

exports.onTap = onTap;
```

Now that you've mastered the *Stack Layout*, you're ready for some more advanced layout techniques. In chapter 6, you'll continue to learn about layout containers. I'll also continue to build upon the Tekmo app, so keep reading!

5.3 Summary

In this chapter, you learned that:

- *Stack Layouts* allow you to organize UI elements by *stacking* them on top of each other
- Although *Text Fields* and *Text Views* are both textboxes, *Text Fields* are single-line textboxes and *Text Views* are multi-line text boxes
- A *Scroll View* can be used to add a scroll bar to a page's UI.
- To debug your app, you should use the *Console* module to log messages and data to the terminal or command prompt.

5.4 Exercise

In this chapter, you learned how to organize UI elements on the screen with various layouts. Try using what you've learned to do the following:

- Using a *Stack Layout*, place three paragraphs of text on a page. Don't let the text scroll off the side of the page.

- Enable scrolling on the page you created in the previous exercise step.
- Using the page you created in the previous exercise steps, display a Dialog alert when the page is loaded.

5.5 Solutions

To place three paragraphs of text on a page within a Stack Layout while ensuring the text does not scroll off the side of the page, use the code listed in listing 5.14.

Listing 5.14 Solution to placing three paragraphs of text within a Stack Layout

```
<StackLayout>
    <Label textWrap="true" text="paragraph 1 goes here" />
    <Label textWrap="true" text="paragraph 2 goes here" />
    <Label textWrap="true" text="paragraph 3 goes here" />
</StackLayout>
```

To enable scrolling for the previous exercise, nest the Stack Layout element beneath a `<ScrollView>` element.

To display a Dialog alert when the page is loaded, add `"loaded="onLoaded"` to the Page element, then add JavaScript to the `onLoaded()` function of the page in the page's .js file. Listing 5.15 shows a complete solution for the page XML of all exercises.

Listing 5.15 Complete solution with three stacked paragraphs of text, scrolling enabled, and loading event wired to the `onLoaded()` JavaScript function

```
<Page loaded="onLoaded">
    <ScrollView>
        <StackLayout>
            <Label textWrap="true" text="paragraph 1 goes here" />
            <Label textWrap="true" text="paragraph 2 goes here" />
            <Label textWrap="true" text="paragraph 3 goes here" />
        </StackLayout>
    </ScrollView>
</Page>
```

Listing 5.16 shows the corresponding page's .js file contents to display the Dialog alert upon page load.

Listing 5.16 JavaScript file contents to display a Dialog alert upon page load

```
var dialogsModule = require("ui/dialogs");

function onLoaded(args) {
    dialogsModule.alert("Page loaded.");
}

exports.onLoaded = onLoaded;
```

6

Using advanced layouts

This chapter covers

- How to organize your app user interface (UI) with the grid layout
- How layout containers can be nested underneath each other to make more advanced UI designs
- How a grid layout can be used to create uniformly-sized rows and columns
- The three different ways in which you can specify the size (width and height) of grid layout rows and columns

In Chapter 5, you learned about the stack layout and how it allows you to stack UI components on top of each other. This layout is a foundational layout container that will appear every app you write; however, it is limited. In this chapter, you will be introduced to an additional layout: the grid layout. After you've learned about the grid layout, you'll be able to construct more advanced and complex UI designs. In fact, with just the stack and grid layouts, you'll be able to create pretty much any UI. That may seem to be a stretch, but in our experience, we've been able to construct about 95% of our app layouts by using these two layouts.

6.1 *Introducing the grid layout*

Imagine you are developing a mobile app with a series of labels and text fields arranged in a series of rows and column. The profile page in figure 6.1 shows an example of one such UI.

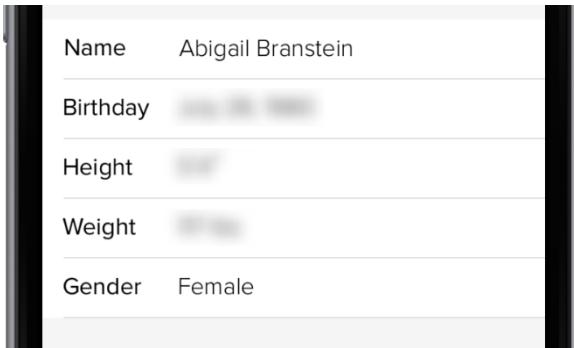


Figure 6.1 A mobile app profile page with labels and textboxes organized in rows and columns.

With the stack layout, you already know how to create a UI with various elements stacked on top of each other, but stack layouts aren't the right choice for UIs that can easily conform to a grid. The grid layout is a perfect fit!

DEFINITION The grid layout is a layout container with the ability to organize UI elements into a collection of rows and columns.

Listing 6.1 shows the XML code used to display a grid layout on the screen.

Listing 6.1 Adding a grid layout to a page

```
<Page>
  <GridLayout rows="*, *, *" columns="*, *"
    width="300" height="450">          #A
    ...
  </GridLayout>                      #B
</Page>
#A This creates a grid layout with three rows and two columns. Don't worry about the rows and columns syntax right now, I'll cover that later.
#B The grid is forced to 300 pixels wide by 450 pixels tall
#C Nested UI controls go here.
```

To add a grid layout to a page, you add the `<GridLayout rows="..." columns="...">` element. You will also notice the `width="300"` and `height="450"` properties. You haven't learned about these properties yet, but they are named well and self-explanatory. Width and height describe their exact purpose: to set the layout's width and height to a given number of pixels.

As you can see, the grid layout element syntax is simple; however, when you define the layout, you also need to include the *rows* and *columns* properties.

DEFINITION The *rows* and *columns* properties of the grid layout define the number of rows and columns within the layout container.

In listing 6.1, we created a grid layout with three rows and two columns. You can tell this by looking at the number of comma-separated values within each of the properties. The rows property has three comma-separated values (`rows="*, *, *"`), meaning the grid layout has three rows. The columns property has two comma-separated values (`columns="*, *"`), meaning the grid layout has two rows. Right now, it's not important that you understand what these comma-separated values mean. We'll cover that later in this chapter. Instead, just remember that for each comma-separated value, there is a row or column.

6.2 Adding content to a grid layout

In the previous section, you learned how to create a *grid layout* with three rows and two columns. Before we teach you how to organize UI element inside a *grid layout*, let's visualize a grid three rows tall and 2 columns wide (figure 6.2).

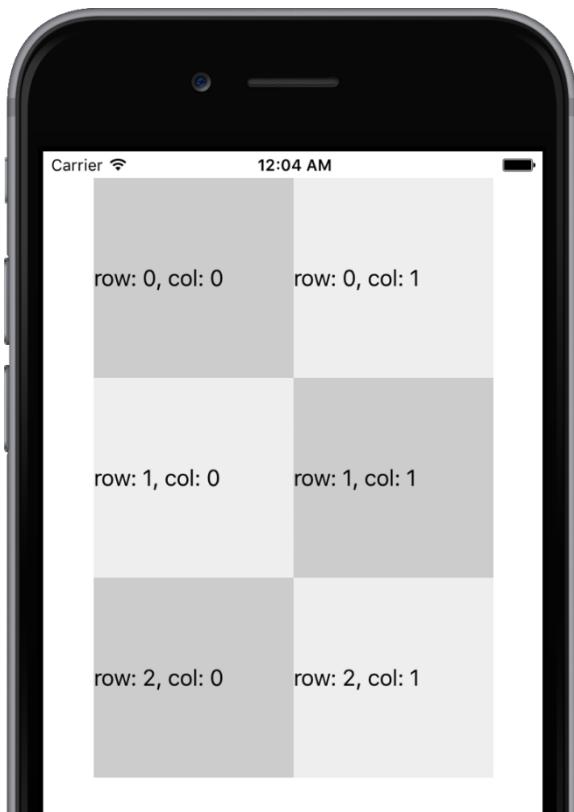


Figure 6.2 A grid consisting of 3 rows and 2 columns.

A *grid layout* of three rows and two columns has a total of six *cells*, with *cell coordinates* of (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), and (2, 1).

DEFINITION A cell refers to the intersection of a specific row and column within a grid. Cells are the natural byproduct of creating a grid. The cell terminology is not specific to NativeScript, but instead a universally accepted term when speaking about a particular row and column of a grid. Just like pixels on a screen, cells can be referred to using a coordinate notation (row #, column #). Using this notation, a grid with two rows and two columns, there are a total of four cells: the upper left cell (0, 0), the upper right cell (0, 1), the lower left cell (1, 0), and the lower right cell (1, 1).

As you learn how to add UI elements to a grid layout, be sure to keep the cell coordinates of each cell in our 3×2 grid in mind.

Just like the stack layout, the grid layout is a container, meaning UI elements nested inside of the layout are governed by the rules of the layout. The grid layout's rules are easy to remember because there is only one rule: each UI element must identify the cell in which the component will appear.

6.2.1 Adding a single UI component to a grid cell

Using this rule, let's add a *products* page to the Tekmo app, then add a grid layout to the page. You'll recall from an earlier chapter that a page consists of an XML, JavaScript, and CSS file. For now, we're just going to add the XML file, naming it *products.xml*. Place the *products.xml* page into the *views/products* folder of your app, as shown in figure 6.3.

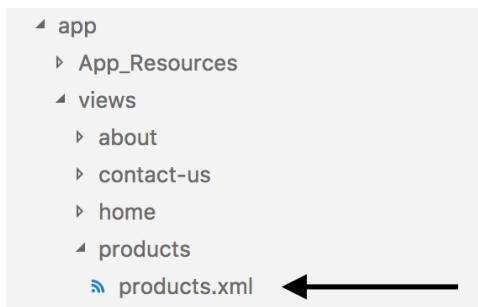


Figure 6.3 The *products.xml* file has been added to the *views/products* folder of the Tekmo app

Once we're finished with the products page, it will display the six vintage video games Tekmo sells: Couch Commander, Mummy Madness, Pyro Robots, Rescue Pups, Super Marshmallow Man, and Vampire Valkyrie. Let's start by adding a *Page* and *GridLayout* element to the *products.xml* file. The grid will be 3×2 (three rows by two columns). In each grid cell, we'll place a *Label* element, representing each game's title (listing 6.2 and figure 6.4).

Listing 6.2 A three-row by two-column grid layout with a label in each cell

<Page>

```
<GridLayout rows="*,*,*" columns="*,*" width="300" height="450">
  <Label row="0" col="0" text="Couch Commander" textWrap="true"
    style="background-color: #CCCCCC" /> #A
  <Label row="0" col="1" text="Mummy Madness" textWrap="true"
    style="background-color: #EEEEEE" /> #A
  <Label row="1" col="0" text="Pyro Robots" textWrap="true"
    style="background-color: #EEEEEE" /> #A
  <Label row="1" col="1" text="Rescue Pups" textWrap="true"
    style="background-color: #CCCCCC" /> #A
  <Label row="2" col="0" text="Super Marshmallow Man"
    textWrap="true" style="background-color: #CCCCCC" /> #A
  <Label row="2" col="1" text="Vampire Valkyrie" textWrap="true"
    style="background-color: #EEEEEE" /> #A
</GridLayout>
</Page>
```

#A Each cell has a Label contained within. A row and col property are added to each Label to tell the GridLayout where to place the component. Grid cell coordinates are always zero-based numbers.



Figure 6.4 A grid layout with a Label in each cell. The Label text indicates the grid cell coordinate at which the Label is placed. Note that each cell has been shaded to better visualize the boundaries.

When adding a UI element to a *grid layout*, you must specify the grid cell coordinates at which the component will be placed. This is done by adding a `row="row #"` and `col="column #"` property to the component. Listing 6.2 shows six Label elements added to the grid layout. To place the Mummy Madness label in the first row and second column (grid cell coordinates 0, 1), you add `row="0"` and `col="1"` to the Label XML element.

WARNING Grid cell coordinates are always zero-based, meaning (0, 0) refers to the first row and first column. This can be confusing if you're not used to thinking in zero-based numbers or are used to programming in a language that doesn't use zero-based indexes. Watch out!

You may have noticed the `style="background-color: ..."` attribute added to each label in listing 6.2. It looks just like an HTML tag for a CSS style property to change the background color of each grid cell. Well, that's exactly what it does. But wait, this isn't HTML, it's NativeScript! We're not really ready to talk about CSS yet, so we're going to skip by the details (and explanation) right now. Don't worry, though. We'll cover this in the next chapter extensively. For now, just ignore the style attributes.

6.2.2 Adding multiple UI elements to a grid cell

In some circumstances, adding a single UI control to a grid cell is sufficient. Complex UIs may call for you to add two, three, or even more components into a cell. For example, what if we wanted to add an image and price for each game displayed on the Tekmo product page? At first, you may think to add these additional components to the grid layout and give them each a row and column value. Unfortunately, this won't work. But why?

WARNING You can assign only a single UI component to a grid cell.

Although we haven't specifically pointed out how to add multiple items to a grid cell, you've already learned everything you need to know. Back in chapter 5, you learned how a stack layout can be used to add multiple UI elements to a page. Stack layouts (and other layouts) are just UI elements that allow you to organize other UI elements on the screen.

Let's get back to the original question, "How do you add multiple UI components to a grid cell?" The answer is to use another layout container (like a stack layout) in each grid cell.

TIP To add multiple UI elements into a grid cell, use another layout container.

Let's apply what you just learned about nesting layout containers and modify the Tekmo app's products page. In listing 6.3 and figure 6.5, we've added an image and price for each product to the grid layout.

Listing 6.3 Using nested layouts to add multiple components to a grid cell

```
<GridLayout rows="*,*,*" columns="*,*" width="300"      #A
            height="450">
    <StackLayout row="0" col="0" style="background-color: #CCCCCC"> #B
        <Label text="Couch Commander" textWrap="true" />           #C
        <Image src="res://couchcommander" width="75" height="75" />
```

```
<Label text="$24.99" /> #C
</StackLayout>
<StackLayout row="0" col="1" style="background-color: #EEEEEE">
    <Label text="Mummy Madness" textWrap="true" />
    <Image src="res://mummymadness" width="75" height="75" />
    <Label text="$32.99" />
</StackLayout>
...
</GridLayout>
#A The GridLayout definition doesn't change when nesting multiple components
#B A nested layout is added with the row="0" and col="0" properties previously included on the Label
#C Note the components nested under the Stack Layout do not need a row and col property definition because
the Stack Layout has already defined the values
#D Remaining grid cell component definitions go here, following the same pattern
```

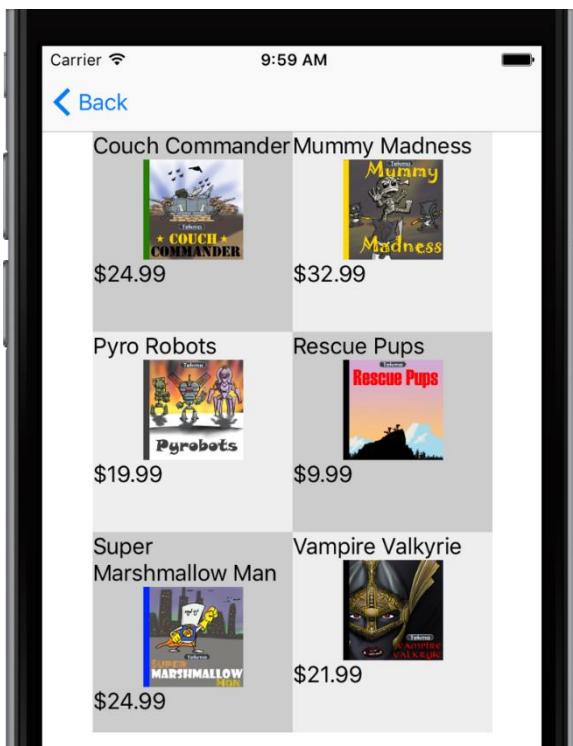


Figure 6.5 A nested stack layout has been added to the grid layout. The video game title, image, and price were then added to the stack layout, allowing all three components to be included within a grid cell.

There's a lot going on in the product page's XML code, so we'll break it down step by step, starting with the `GridLayout` element. When changing this code to include multiple components per grid cell, the element for the grid layout doesn't change.

Beneath the grid layout we added a series of stack layout elements. You'll recall from earlier in this chapter that the tip for including multiple components within a grid cell is to nest another layout container inside of the grid layout. The `<StackLayout row="0" col="0">` XML code is our first nested layout. Within the element's opening tag, you'll see the location within the grid has been identified by including `row="0"` and `col="0"`. This particular stack layout will be in the first row and first column.

The next several lines of XML code in listing 6.3 are the product title, image, and price. The code used for the title and price should look familiar because you've already learned about labels. But, there is a subtle change we made to each of the title Label elements. Take the Couch Commander title as an example. Before adding the nested stack layouts, the title label was `<Label text="Couch Commander" textWrap="true" row="0" col="0" />`. Afterward, the label was `<Label text="Couch Commander" textWrap="true" />`. When I moved the label inside of the Stack Layout, I removed the `row="0"` and `col="0"` properties and placed them on the Stack Layout element. Why?

NOTE The grid layout allows you to have only one nested XML element assigned to each grid cell. When you nest another layout container beneath a grid layout, the nested layout is the only XML element that needs to be assigned to a grid cell with the `row` and `col` properties. Child elements of the nested layout do not need to specify these properties.

The final pieces of code we want to highlight in listing 6.3 are the image elements within each stack layout. For example, the Couch Commander image element is `<Image src="res://couch-commander" width="75" height="75" />`. Although you haven't learned about the `Image` element yet, you can probably guess what it does.

DEFINITION The `Image` element displays an image on the screen. You can set the width and height of the displayed image with using the `width` and `height` attributes.

Right now, we don't want to explain images in detail, so just remember that `Image` elements display a product image on the screen when the app runs. We will be going into the `Image` element in more detail in a later chapter.

6.2.3 *Spanning UI elements across multiple rows and columns*

So far, you've learned how to display UI elements within grid cells, but each element occupied a single grid cell. It's possible for UI elements to occupy multiple grid cells at a time. Let's take the Tekmo Product page as an example. The current page has a grid of six products (figure 6.6), with each product occupying a single grid cell.

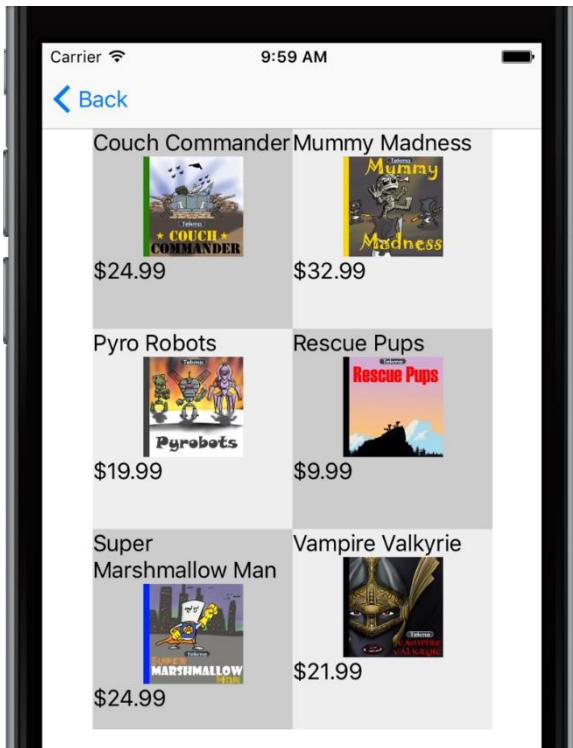


Figure 6.6 Tekmo's product listing, with each of the six products occupying a single grid cell.

Suppose Tekmo wanted to promote a product and have the product's title, image, price, and a brief description of the product displayed more prominently? Wouldn't it be nice to take a featured product and stretch its content across the entire first row of the grid? You can stretch elements across rows and columns, and it's called *spanning*.

DEFINITION Spanning refers to extending a UI element across multiple rows and columns. The span of a UI element refers to the number of grid cells (rows and columns) the element occupies. UI elements can independently span any number of rows and column within a Grid Layout. By default, their span is a single row and column.

Spanning an element is easy, and uses the `rowSpan` and `colSpan` properties to define the number of rows and columns an element occupies. Listing 6.4 and figure 6.7 show how the Super Marshmallow Man game is featured on the product page by spanning it across the first row.

Listing 6.4 Featuring a product by spanning it across multiple grid layout columns

```
<GridLayout rows="*,*,*,*" columns="*,*" width="300"      #A  
           height="600">                                #A
```

```

<StackLayout row="0" col="0" rowSpan="1" colSpan="2"      #B
    style="background-color: #DDDDDD />                      #B
    <Label text="Super Marshmallow Man" textWrap="true" /> #C
    <Image src="res://super-marshmallow-man"               #C
        width="75" height="75" />                          #C
    <Label text="Escape from certain death in..." />     #C
    <Label text="On Sale! $14.99" />                        #C
</StackLayout>

<StackLayout row="1" col="0" style="background-color: #CCCCCC"> #D
    <Label text="Couch Commander" textWrap="true"           #D
        width="75" height="75" />                          #D
    <Image src="res://couch-commander" />                 #D
    <Label text="$24.99" />                            #D
</StackLayout>

<StackLayout row="1" col="1" style="background-color: #EEEEEE"> #D
    <Label text="Mummy Madness" textWrap="true"           #D
    <Image src="res://mummy-madness"                     #D
        width="75" height="75" />                          #D
    <Label text="$32.99" />                            #D
</StackLayout>

...
#E

</GridLayout>
#A When one of the products is featured, it will occupy an entire row, requiring four rows of product to display everything. Notice the additional row in the rows property.
#B Starting at grid cell (0, 0), this element spans one row and two columns
#C Additional content is added to highlight the featured product
#D The remaining product row and col properties are adjusted to account for the first product occupying more space
#E The remaining products go here

```

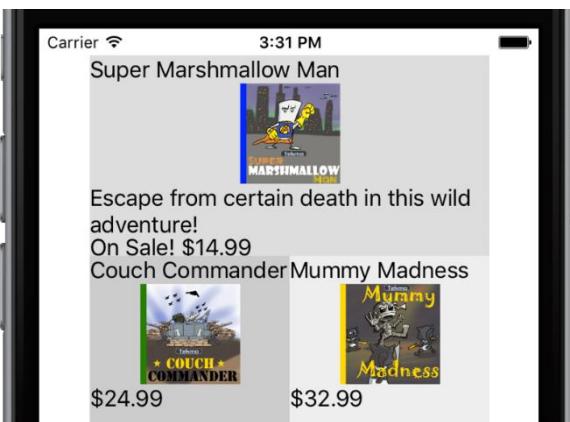


Figure 6.7 The Super Marshmallow Man game has been featured and spans two columns in the first row.

Spanning UI elements across multiple rows and columns is easy. On the product page, the Super Marshmallow Man game has been featured by spanning it across two columns. When the Super Marshmallow Man's stack layout is spanned across two columns, the `row="0"` and `col="0"` properties indicate that the UI element should start at the first row and first column. From that starting point, the element should span one row and two columns, as defined by `rowSpan="1"` and `colSpan="2"`.

WARNING When you first start spanning UI elements across multiple rows and columns, it may be easy to mentally visualize where different elements are displayed. But this can quickly become confusing when you have multiple sets of elements separately spanning rows and columns. If you get confused, don't panic. Put your keyboard down and pick up your pencil and paper. Draw a grid with the same number of rows and columns within your app and lay out your UI elements by hand. Later in this chapter, we'll give you more tips on how to plan your UI. Just remember that simple and low-tech can be powerful allies.

6.3 Controlling grid layout rows and columns

Throughout this chapter, we've purposefully shied away from teaching you how to specify the number of rows and columns in a grid layout. Now that you know the basics of organizing UI elements with the `row` and `col` properties, it's time to dig into this topic. But, before we jump in, we want to let you know that specifying the number of rows and columns can be confusing. We'll take it slow, and give you examples along the way, so be patient.

Let's revisit the basic row and column definition syntax you've already seen. For example, a grid layout defined with `<GridLayout rows="*, *, *, *" columns="*, *">` has four rows and two columns because each comma-separated value within the `rows` and `columns` properties corresponds to a discrete row or column. Said in another way, the number of comma-separated values *is* the number of rows or columns.

TIP To add rows or columns to a grid layout, add comma-separated values to the `rows` or `columns` property. A grid layout with 4 rows will have 4 comma-separated values: `rows="*, *, *, *"`. If you remove a comma-separated value, the grid layout will have 3 rows: `rows="*, *, *"`.

That's not too confusing, right? If this were the only thing the `rows` and `columns` properties did, I'd be right. But, you've only learned half of what the `rows` and `columns` properties control. In addition to specifying the *number* of rows and columns, these properties also describe the *size* (width and height) of each row and column.

NOTE The comma-separated values of the `rows` and `columns` property describe both the number and size of each row and column.

Each comma-separated value has a special meaning and describes how the size of a row or column is determined. In the grid layout examples earlier in this, you've seen a `*` value used for each row and column. The `*` value is one of three special values that can be used to describe the size of rows and columns. The other two types of values are an integer and the word `auto`.

These three methods of specifying the size of rows and columns are what can make understanding grid layouts challenging. But don't worry because we'll take it slow and explain each as we use it on the products page. If you're in a hurry, and want an abridged course in row and column sizing methods, the NativeScript documentation is a great resource. You can find the grid layout documentation at <http://docs.nativescript.org/ui/layout-containers#gridlayout>.

Let's get started by looking at table 6.1, which outlines the three sizing methods and how the value is used to determine the width and height of rows and columns.

Table 6.1 Different methods for specifying the size of a row or column in a grid layout.

Sizing Method	Value	Description
pixel	# of pixels	Sets the width and height of a row or column by specifying an exact (or fixed) number of pixels.
percentage	*	Distributes rows and columns across all available space in the grid layout. Available space is proportionately divided across all percentage rows and columns. Percentage sizing always uses the maximum amount of space available in a grid layout.
automatic	auto	Sets the size of rows and columns based on the size of UI elements within the grid cells. Row height is set to the height of the tallest grid cell in the row. Column width is set to the width of the widest grid cell in the column. Automatic sizing uses the minimum amount of space needed by its contents.

The pixel sizing method is the simplest of the three sizing methods: by specifying an integer as the sizing value, you can set the height and width of rows and columns. You can check out the official NativeScript documentation at <http://docs.nativescript.org/ui/layout-containers#gridlayout> to find out more.

6.3.1 Understanding percentage sizing in the Tekmo app

When we last updated the Tekmo app, we had added a fourth row to the grid layout so the Super Marshmallow Man game could be highlighted at the top of the page (listing 6.5).

Listing 6.5 Products page with four rows, highlighting Super Marshmallow Man in the first row

```
<GridLayout rows="*, *, *, *" columns="*, *" width="300"      #A
           height="600">                                     #A

  <StackLayout row="0" col="0" rowSpan="1" colSpan="2"
               style="background-color: #DDDDDD" />

  <Label text="Super Marshmallow Man" textWrap="true" />
  <Image src="res://super-marshmallow-man" />
  <Label text="Escape from certain death in..." />
  <Label text="On Sale! $14.99" />
</StackLayout>
```

```
...  
</GridLayout>  
#A The grid layout uses percentage sizing for the 4 rows and 2 columns
```

The grid layout on the products page uses the percentage sizing method for each of the 4 rows and 2 columns. But what does that really mean?

DEFINITION Percentage sizing is a method for determining the size of rows and columns by proportionately distributing available grid space across each row or column using the method.

WARNING Before we go any further, we want you to know that percentage sizing can be confusing, because it's not intuitive (at least to us). We think an intuitive percentage-based sizing syntax would include percentage signs. For example, `rows="25%, 25%, 25%, 25%"` to indicate there are 4 rows, each with 25% of the space. But, that's not how NativeScript works. Instead, you must use the star syntax.

Because we feel percentage sizing is confusing, we've spent a little more time to carefully explain how percentage sizing works. We've also focused on columns exclusively in this section and purposefully ignored how percentage sizing applies to rows (because they work identically).

GRID SPACE VERSUS AVAILABLE GRID SPACE

To fully understand percentage sizing, it's important that you understand the difference between *grid space* and *available grid space*. When NativeScript displays a grid on a page, it first determines the total size the grid can occupy on a page. The total size is referred to as the *grid space*.

DEFINITION Grid space is the total size a grid occupies on a page.

After the grid space is determined, NativeScript allocates space from the grid space to rows and columns using the pixel and automatic sizing methods. After these columns have been given a portion of the grid space, the remaining space is referred to as the *available grid space*.

DEFINITION Available grid space is the width and height remaining out of the grid space after pixel and automatic columns have been displayed.

For now, we'll be using examples where the grid space and available grid space are the same: the entire width and height of the grid. Later, we'll incorporate a more complex example.

APPLYING PERCENTAGE SIZING TO THE TEKMO APP

The easiest example of percentage sizing is a grid with a single column, for example `columns="*"`. As you've learned, percentage sizing proportionately distributes the percentage columns across the available grid space. The available grid space is 100%. This results in a single column being 100% of the grid's width.

Now, let's look at the Tekmo app's product page with two columns using percentage sizing: `columns="*, *"`. You'll also recall the grid layout is set to a width of 300 pixels with the width property: `width="300"`. Based on the columns and width properties, you know the available grid space is 100%, or 300 pixels. With two percentage-sized columns, each column will receive 50% of the grid's width, or 150 pixels. Figure 6.8 shows the two product page columns, each consuming 50% of the grid's width.

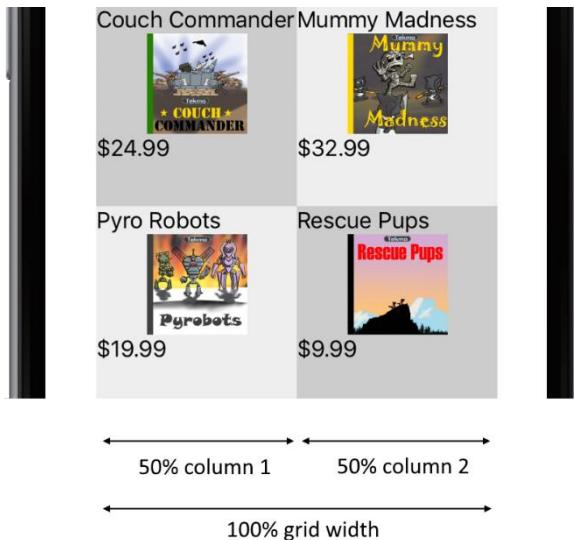


Figure 6.8 The product page showing two percentage size columns, each consuming 50% of the grid's available column space.

Now that you have a handle on column sizing, let's translate this to rows. The products page has four rows and a height of 600 pixels defined in the grid layout: `<GridLayout rows="*, *, *, *" height="600">`. The available grid space for rows is 100%, or 600 pixels. With 4 rows, each row will receive 25% of the grid's height, or 150 pixels. Figure 6.9 shows how the four rows are evenly sized.

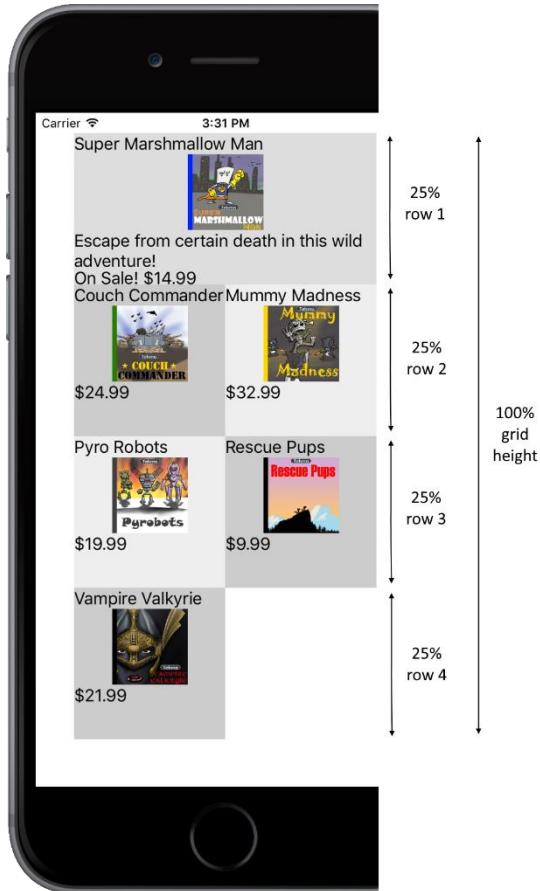


Figure 6.9 The product page showing 4 percentage size rows, each consuming 25% of the grid's available row space, or 150 pixels.

Tekmo's product page rows and column sized with the percentage method are basic examples of how the percentage method works. If you're interested in more complex scenarios, look at the *percentage sizing deep dive* sidebar in this chapter, or check out the NativeScript documentation at <https://docs.nativescript.org/ui/layout-containers#gridlayout>.

Percentage sizing deep dive

You've already seen basic examples of the percentage sizing, where each column is sized the same, such as `columns="*, *"`. Determining the percentage of the available grid space is easy in these circumstances, so I didn't bother to paint the full picture. Here's what's really happening. When

NativeScript displays a grid layout with columns defined using the percentage sizing convention, the resulting column width is determined using a *calculation*.

To calculate the width of columns using the percentage method, NativeScript adds the total number of stars together, then divides each column's star count by this total. The resulting dividend is converted to a percentage. For example, `column="*, *, *"` has 3 stars. The first column's width is calculated by dividing 1 by 3 and converting the result into a percentage (33.3%). 1 is divided by 3 because there is 1 star defined for the first column, and a total of 3 stars across all percentage-sized columns. The same calculation is done for the second and third columns.

Understanding the percentage calculation may seem trivial because you've been learning about grids with equally-sized columns. But, it's possible for grid columns to be sized differently by placing a number in front of each *. For example, `columns="2*, *"` defines 2 columns, but column 1 (66.6%) is proportionately twice the size of column 2, which is 33.3%.

Let's go back to the percentage calculation to understand why the first column is 66.6% and the second column is 33.3%. You'll recall the calculation adds the total number of stars in all columns, which is 3 stars (because $2*$ means 2 stars and $2* + * = 3$ stars). Each column's star count is divided by the total number of stars and changed to a percentage. This means the first column's width is $2*$ (2 stars) divided by $3*$ (3 stars) = 66.6% and the second column's width is $*$ (1 star) divided by $3*$ (3 stars) = 33.3%.

As you can see, the percentage method's calculation can get confusing. If you do get confused bookmark this page, keeping this sidebar as a point of reference.

6.3.2 Using automatic sizing for rows and columns

In the previous section, you learned how the Tekmo app's products page used the percentage method to create 4 rows and 2 columns, all of equal width and height. The percentage method is a great choice when you want multiple columns or rows to be sized proportionately to each other.

What happens when a UI element doesn't fit just right within the equally sized rows and columns? For example, let's take the Super Marshmallow Man game, which is highlighted at the top of the products page (figure 6.10).

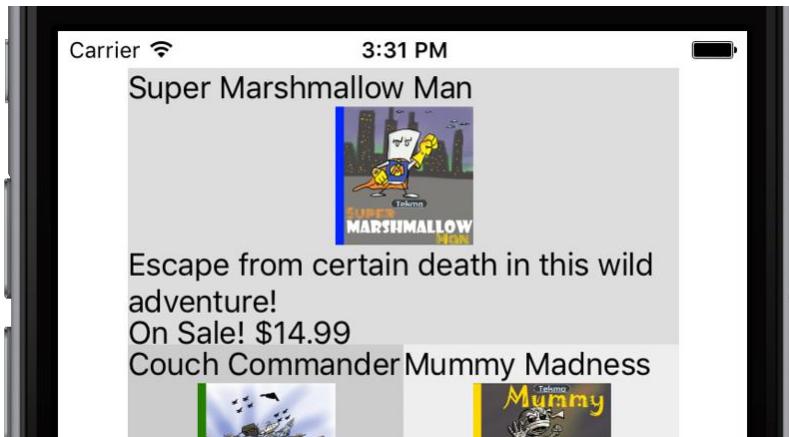


Figure 6.10 The products page highlighting the Super Marshmallow Man game at the top of the page.

Right now, the text "Escape from certain death in this wild adventure" fits nicely on 2 lines without pushing the "On Sale! \$14.99" text out of the first grid row. But what if the description was a little longer, as in figure 6.11.

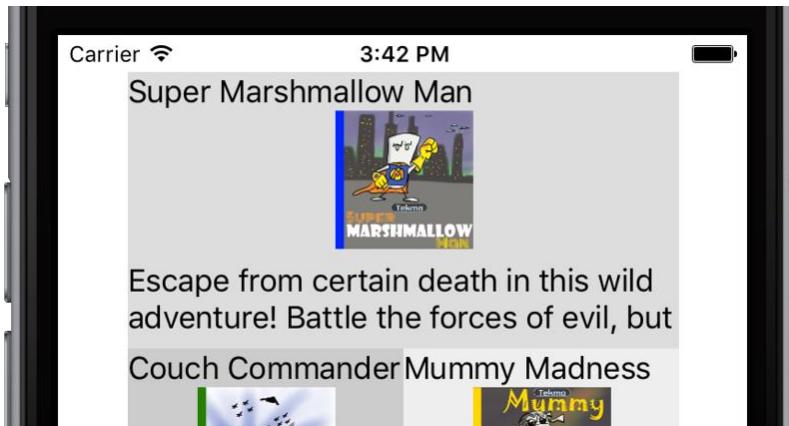


Figure 6.11 The descriptive text of Super Marshmallow Man is more than 2 lines long, hiding the price.

Ideally, we'd want the row containing Super Marshmallow Man to expand and automatically accommodate the lengthier text. Unfortunately, it doesn't because the first row is sized using the percentage method: `rows="*, *, *, *"`, forcing the row to have a height identical to the second, third, and fourth rows. So, what can we do? *Automatic sizing* to the rescue!

DEFINITION The automatic sizing method sets the size of rows and columns based on the size of UI elements within the grid cells. Row height is set to the height of the tallest grid cell in the row. Column width is set to the width of the widest grid cell in the column. Automatic sizing uses the minimum amount of space needed by its contents.

Automatic sizing is a great solution to my problem because it will automatically increase the size of a row to accommodate a longer description for Super Marshmallow Man. First, we'll change the first row's sizing method from the percentage to automatic. We'll do this by changing the rows property of the grid layout from `rows="*, *, *, *"` to `rows="auto, *, *, *"`. Because of the automatic row-sizing change we made, the first row will automatically expand to show the following text added to the description label:

```
<Label text="Escape from certain death in this wild adventure! Battle the forces of evil, but don't melt!" />
```

Figure 6.12 shows the resulting changes, with a larger first row.

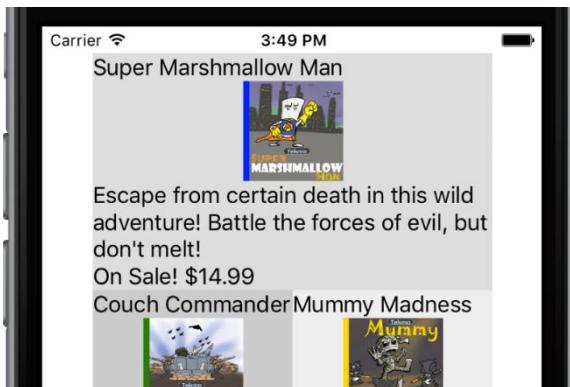


Figure 6.12 The Super Marshmallow Man description has been expanded and the row height set to auto, allowing the row to automatically increase in height.

We've made a lot of changes to the products page throughout this section. If you haven't been able to follow along, you can look at listing 6.6 which contains the final version of the products page.

Listing 6.6 Final version of the products page

```
<Page>
  <GridLayout rows="auto, *, *, *" columns="*, *" width="300"
  height="600">

    <StackLayout row="0" col="0" rowSpan="1" colSpan="2"
    style="background-color: #DDDDDD" >
      <Label text="Super Marshmallow Man" textWrap="true" />
      <Image src="res://super-marshmallow-man" />
      <Label text="Escape from certain death in this wild adventure!
      Battle the forces of evil, but don't melt!" textWrap="true" />
      <Label text="On Sale! $14.99" />
```

```
</StackLayout>

<StackLayout row="1" col="0" style="background-color: #CCCCCC;">
  <Label text="Couch Commander" textWrap="true" />
  <Image src="res://game" width="75" height="75"
    stretch="aspectFill" />
  <Label text="$24.99" />
</StackLayout>

<StackLayout row="1" col="1" style="background-color: #EEEEEE;">
  <Label text="Mummy Madness" textWrap="true" />
  <Image src="res://game" width="75" height="75"
    stretch="aspectFill" />
  <Label text="$32.99" />
</StackLayout>

<StackLayout row="2" col="0" style="background-color: #EEEEEE;">
  <Label text="Pyro Robots" textWrap="true" />
  <Image src="res://game" width="75" height="75"
    stretch="aspectFill" />
  <Label text="$19.99" />
</StackLayout>

<StackLayout row="2" col="1" style="background-color: #CCCCCC;">
  <Label text="Rescue Pups" textWrap="true" />
  <Image src="res://game" width="75" height="75"
    stretch="aspectFill" />
  <Label text="$9.99" />
</StackLayout>

<StackLayout row="3" col="0" style="background-color: #CCCCCC;">
  <Label text="Vampire Valkyrie" textWrap="true" />
  <Image src="res://game" width="75" height="75"
    stretch="aspectFill" />
  <Label text="$21.99" />
</StackLayout>
</GridLayout>
</Page>
```

6.3.3 Additional layout containers

Over the past two chapters, you learned about the two most common layout containers: Stack Layout and Grid Layout. By using these layouts and the practice of nested layouts, you can create complex layouts. But, sometimes you need something with a little more flexibility.

NativeScript has three additional layouts you can use to build more complex (and flexible) UIs: absolute, dock, and wrap layouts. We want to make you aware of these layouts, but we're not going to cover them aside from the mention above. Why? In practice, 95% of the layouts you'll create in a NativeScript app will use the stack and grid layout. In fact, across all the apps we've written, we've only had to use a different layout once. Of course, your mileage may vary, and we don't want to lock you into the stack and grid layouts, but at the same time, don't feel slighted that you've only learned about two layouts.

As you begin developing an app, you can learn how the absolute, dock, and wrap layouts work by reading the official NativeScript documentation at <https://docs.nativescript.org/ui/layout-containers>.

6.4 Summary

In this chapter, you learned that:

- Grid layouts are a collection of rows and columns, allowing you to organize UI elements in a grid
- You define the number and method for sizing rows and column via comma-separated values of the rows and columns property of a grid layout
- There are three methods for specifying the width and height of columns and rows within a grid layout: by pixels, by percentage, and automatically
- Rows and columns using the percentage-sizing method take up all available grid space, which is determined only after rows and columns using the pixel and automatic sizing methods are displayed

6.5 Exercise

In this chapter, you learned how to organize UI elements on the screen with various layouts. Try using what you've learned to do the following:

- Using a grid layout, create a three-row, three-column layout container that is 500 pixels wide and 1000 pixels high. Make the first row 300 pixels high, the second row automatically adjust to the maximum height of the elements in the row, and the third row consume the remaining available grid space. The first, second, and third columns should be 300 pixels, 100 pixels, and 100 pixels wide, respectively. Place a label in each grid cell with the text set to the coordinate for each grid cell.

6.6 Solutions

To complete the grid layout exercise, do the following:

- Define the grid layout with `<GridLayout width="500" height="1000">`.
- Add `rows="300, auto, *"` to the Gird-Layout element.
- Add `columns="300, 100, 100"` to the Grid-Layout element.
- Add the following Labels to the grid layout:
 - `<Label text="(0,0)" row="0" col="0" />`
 - `<Label text="(0,1)" row="0" col="1" />`
 - `<Label text="(0,2)" row="0" col="2" />`
 - `<Label text="(1,0)" row="1" col="0" />`
 - `<Label text="(1,1)" row="1" col="1" />`
 - `<Label text="(1,2)" row="1" col="2" />`
 - `<Label text="(2,0)" row="2" col="0" />`
 - `<Label text="(2,1)" row="2" col="1" />`
 - `<Label text="(2,2)" row="2" col="2" />`

7

Styling NativeScript apps

This chapter covers

- Using CSS to style NativeScript apps
- Displaying images that visually appear the same size on various-sized mobile devices
- Accounting for varying DPI densities when displaying images

Developing mobile apps isn't about just the technical know-how. Mobile app development is also about creating beautiful apps, with a great user experience and stunning visuals. In other words, apps let mobile developers highlight their creative capabilities. Up until this point in the book, I've been focused on creating content and the technical side of organizing the UI; let's be honest: the Tekmo app looks bland. I've ignored making the UI look appealing, and instead focused on functionality. It's time for a change.

For an app to feel highly functional, the UI needs to be visually appealing. Think about it: how many apps are in the Google Play and iTunes stores? Millions of apps with tens of billions of downloads. How do you differentiate your app from the millions of other apps? What makes your app special? Well-designed and thoughtful functionality is one step, and a compelling UI is another.

In this chapter, you'll learn how to use the tools that can transform your app from *wimpy* to *whammy!* But, I won't lie to you: creating beautiful apps isn't simple. It takes planning and some artistic abilities. Truthfully, I don't have these artistic abilities; however, when I see something stunning in an app, I know it. It's often hard to describe why I think it's stunning, but I know it when I see it.

If you're like me, it's time to buddy-up. Find a friend or co-worker that has that artistic flair and who might be willing to sit down with you for an hour or two over a coffee to sketch out some basic concepts. Good visuals and a solid UI generally don't happen by accident, so it may take a few cups of coffee before everything falls into place. Be patient.

Back in chapter 3, I said to be agile. That tip still applies here. Start with a sketch (not a blank computer screen) when you start to create your apps. But don't spend months refining that sketch before you start coding. After all, if you don't start coding, the app will never get into the hands of someone to test. Briefly

plan with a rough idea, then code. Along the way to a functional app, your app's design will take turns you didn't anticipate.

Let's get started!

7.1 Using cascading style sheets

I've been comparing NativeScript app development to HTML application development throughout the beginning of the book, not because it's convenient, but because they're similar. NativeScript app styling is another opportunity for this comparison because you style your apps with cascading style sheets (CSS).

7.1.1 Styling basics

If you recall from earlier chapters, a NativeScript page has three components: an XML file, a JavaScript file, and a CSS file (figure 7.1).

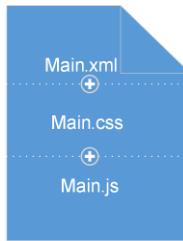


Figure 7.1 Pages have three components: an XML file, a CSS file, and a JavaScript file.

We want to come back to my original statement that you style your NativeScript apps with CSS: there are some exceptions.

NOTE NativeScript apps can be styled with a subset of CSS. In other words, not all aspects of CSS can be used in (or apply to) NativeScript apps.

The CSS Specification

CSS is an incredibly broad term, referring to a collection of various specifications on how to define style (for example, fonts, colors, spacing, and so on), and is intended to apply to HTML. Although NativeScript apps are like HTML apps, they're not the same technology.

A group known as the CSS Working Group meets regularly to discuss and define an official collection of specifications that form what most people commonly refer to as CSS. The specifications are incredibly detailed and are broken into various sub-areas including media queries, a color module, box alignment, scroll snapping, selectors, style attributes, text module, and so on. These specifications are under continual development. You can find out the current state of CSS and these specifications by visiting <https://www.w3.org/Style/CSS/current-work>.

When NativeScript loads a page, these three files are automatically loaded. So, by creating a file named `page-name.css`, you can create page-specific CSS style rules. In addition to creating page-specific CSS styles, you can also specify CSS style for your entire application and on individual UI elements. Table 7.1 describes the three ways of specifying style.

Table 7.1 Different methods and locations for specifying CSS style within an app

Method	Location	Description
global	<code>app.css</code>	Sets CSS style rules for the entire app. All pages will inherit the style rules from the <code>app.css</code> file.
page-specific	<code><page-name>.css</code>	Sets CSS style rules for a specific page. To use this method, create a CSS file with the same name as the page and add CSS style rules to the file.
(inline) UI element-specific	<code>style="..." attribute on a UI element</code>	Sets CSS style rules for an individual UI element. Add a <code>style="..."</code> attribute to any UI element's declaration and include style rules within the attribute value.

You can use any combination of these methods when styling an app, so staying organized is important.

TIP If you're developing an app with others, establishing which of the three methods for defining CSS you'll use is important. Discuss this with your team early in your project, and document your decisions.

When you start adding CSS style rules to your app, it can be confusing which of the three CSS styling methods to use: should it be global, page-specific, or defined on a UI element? If you're unsure, consider the following recommendations:

1. Avoid inline UI element-specific style rules. Even though it's possible to use this method, it is not a generally-accepted good practice. CSS styles are meant to be separate from the UI.
2. Start by placing all style rules in a page-specific CSS file.
3. If you find that you're repeating a style rule on more than one page (or if you suspect the style will be used on more than one page), consider removing the style rule from the page-specific CSS files and create a global style rule in the `app.css` file.

SUPPORTED STYLE SELECTORS AND PROPERTIES

NativeScript supports a subset of the CSS selector syntax. You can use 5 different CSS selectors: type selectors, class selectors, ID selectors, hierarchical selectors, and attribute selectors. You should be familiar with these selectors, so I'm not going to explain them in greater detail. If you'd like a quick refresher, the official NativeScript documentation has a good introduction on the selectors at <http://docs.nativescript.org/ui/styling>.

NOTE You might be thinking why NativeScript doesn't support every CSS selector. Not all CSS properties apply to mobile apps. For example, the `:hover` selector is used to select an element when a mouse hovers over it. This isn't applicable to mobile devices.

Like the subset of CSS selectors supported by NativeScript, you can use only a subset of CSS properties in a NativeScript app. The supported properties include many of the common properties you are already familiar with relating to color, font, background images, text, alignment, spacing (margin and padding), and size (width and height). The complete list of supported properties is included online at <http://docs.nativescript.org/ui/styling>.

NOTE Just like selectors, not all properties apply to mobile devices. For example, the `nav-{up|down|left|right}` properties apply to keyboard navigation, so these aren't supported by NativeScript.

7.1.2 Using global CSS styles

Let's start by adding some CSS style rules to the Tekmo app. We'll start by making several global style changes to the app.

NOTE Remember that global CSS changes will apply to the entire app, so these rules should target styling aspects we expect to apply to more than one page.

Most of the app pages have title text that should stand apart from other text on the pages. Create a style rule for title text by adding a CSS class selector to the app.css file. Listing 7.1 shows how to add the selector to the app.css file, specify a font size of 30, center alignment, and a margin of 20 on all sides. A subtitle selector is also specified, giving subtitles a font size of 20.

Listing 7.1 Adding a class selector and properties to style text on all app pages

```
.title {  
    font-size: 30;  
    horizontal-align: center;  
    margin: 20;  
}  
.sub-title {  
    font-size: 20;  
}
```

After adding the global `.title` class selector, you'll need to add the `class="title"` attribute to the title label elements on the Home, About, and Products pages. For example, the Home page's title label will be `<Label textWrap="true" text="Welcome to Tekmo!" class="title" />` after you've added the `class` attribute. The About page also has several labels acting as subtitles. Change the two subtitle labels on the About page to include the `sub-title` class name.

```
<Label text="Our Mission" class="sub-title" />  
<Label text="History" class="sub-title" />
```

Let's look at how this has changed your app. Figure 7.2 shows a before and after snapshot of the About page.

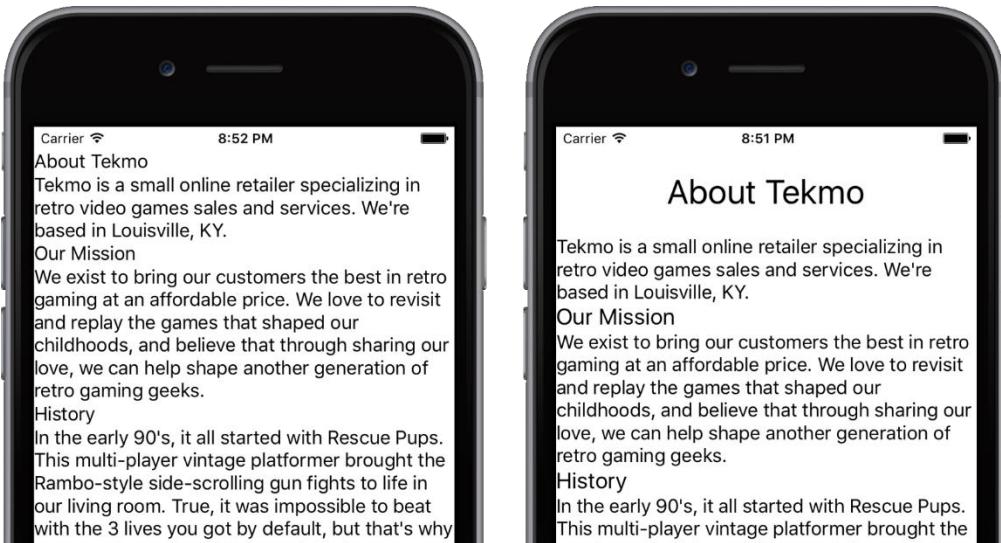


Figure 7.2 The About page without styling (on left) and with styling (on right).

To go along with the changes to the title, provide a default style for all labels, so they have a little more space around the elements (listing 7.2). This is a global style, so it should be added to the app.css file.

Listing 7.2 A type selector for applying margin space around all labels and centering buttons

```
Label {  
    margin-left: 10;  
    margin-right: 10;  
    margin-bottom: 10;  
}
```

Finally, let's reduce the size of the buttons on the Home page, creating a universally-sized button throughout the app (42). You'll recall from chapter 3 that we used the default NativeScript app template to create the Tekmo app. Because we used this template, there's already a button type selector included in the app.css file, as seen in listing 7.3.

WARNING We've relied on the default app template that has a button type selector already, but the default template could change over time. Don't worry – if the default template has changed and your app.css file doesn't have the button selector included, you can add it.

Listing 7.3 The default button type selector in the app.css file

```
button {  
    font-size: 42;  
    horizontal-align: center;  
}
```

This default code styles buttons at a size that is way too big for our purposes. Let's update the button style by removing the `font-size: 42;` property.

With these latest additions, the Home and About pages, as seen in figure 7.3, look a little more presentable.

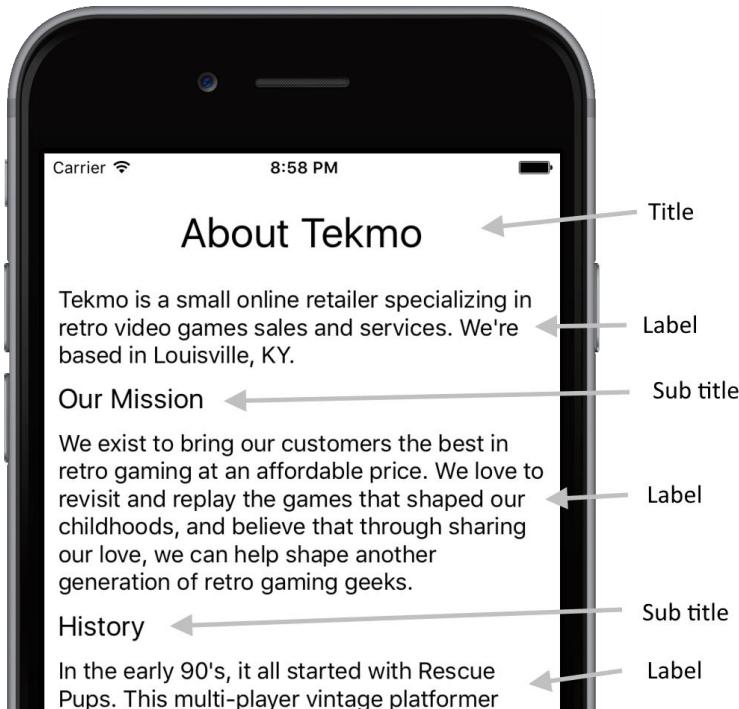


Figure 7.3 The About page with limited text styling.

Now that you've learned some of the basics of styling apps with CSS, let's move on to styling the grid layout on the Products page.

7.1.3 Styling a grid layout with page-specific CSS

Before we start styling the Products page grid layout, let's look at the current state of the page. Figure 7.4 shows the Products page, as of the end of chapter 6.

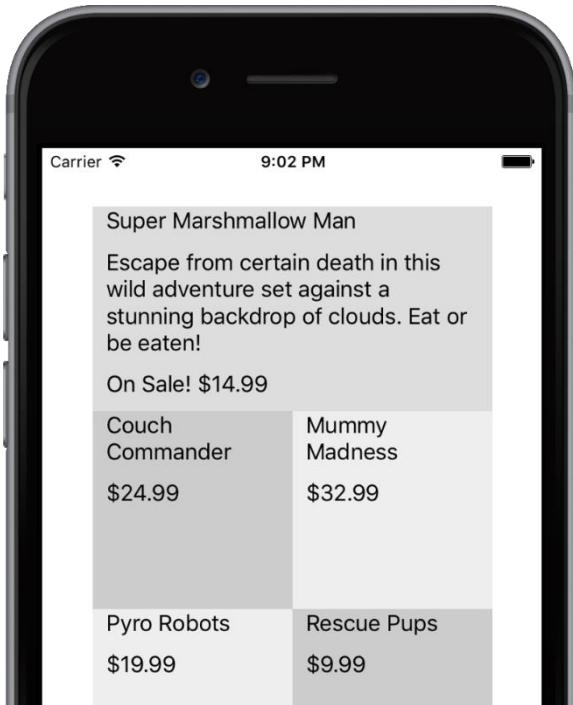


Figure 7.4 The Products page with limited styling, as seen at the end of Chapter 6.

As you can tell, the Product page is fairly plain. Let's make several changes:

- Give all app pages a default background color instead of white.
- Remove the alternating grid cell colors.
- Color the grid cells white and add spacing between them, so an app user can visually tell where one tile ends and another begins.
- Add a solid band of color (a.k.a. title banner) across the top of each grid cell to surround the game title.
- Style the game titles to stand out against the solid band of color.
- Right-align the price and add some color.
- Style the highlighted Super Marshmallow Man grid cell to make it stand out in comparison from other tiles.

This is a lot of change to make at once, so we'll walk through them together. But if you like challenge, go ahead and try it out on your own. As we work through these changes, I won't give you a figure for each step, but you'll see in-progress figures for the major milestones.

SETTING AN APP-WIDE BACKGROUND COLOR

Let's start by giving all the app pages a default background in the app.css file. Listing 7.4 shows the additions.

Listing 7.4 Additions to the app.css file to change the app's background color to a shade of light grey

```
Page {  
    background-color: #EFEFEF;  
}
```

REMOVING INLINE STYLES FROM GRID CELLS

Next, let's remove the inline style attributes (`style="background-text: ..."`) from the grid cells on the Products page. Once you've removed these attributes, your grid layout should match the code in listing 7.5.

Listing 7.5 Resulting file after removing inline style attributes

```
<GridLayout rows="*,*,*,*" columns="*,*" width="300" height="600">  
    <StackLayout row="0" col="0" colSpan="2">  
        <Label text="Super Marshmallow Man" textWrap="true" />  
        <Label textWrap="true" text="Escape from certain death  
            in this wild adventure!" />  
        <Label text="$34.99" />  
    </StackLayout>  
    <StackLayout row="1" col="0">  
        <Label text="Couch Commander" textWrap="true" />  
        <Label text="$24.99" />  
    </StackLayout>  
    <StackLayout row="1" col="1">  
        <Label text="Mummy Madness" textWrap="true" />  
        <Label text="$32.99" />  
    </StackLayout>  
    <StackLayout row="2" col="0">  
        <Label text="Pyro Robots" textWrap="true" />  
        <Label text="$19.99" />  
    </StackLayout>  
    <StackLayout row="2" col="1">  
        <Label text="Rescue Pups" textWrap="true" />  
        <Label text="$9.99" />  
    </StackLayout>  
    <StackLayout row="3" col="0">  
        <Label text="Vampire Valkyrie" textWrap="true" />  
        <Label text="$21.99" />  
    </StackLayout>  
</GridLayout>
```

After making these changes, the Products page looks a bit more "bleh", as shown in figure 7.5. That's ok though; we're ready to start giving it some flair.

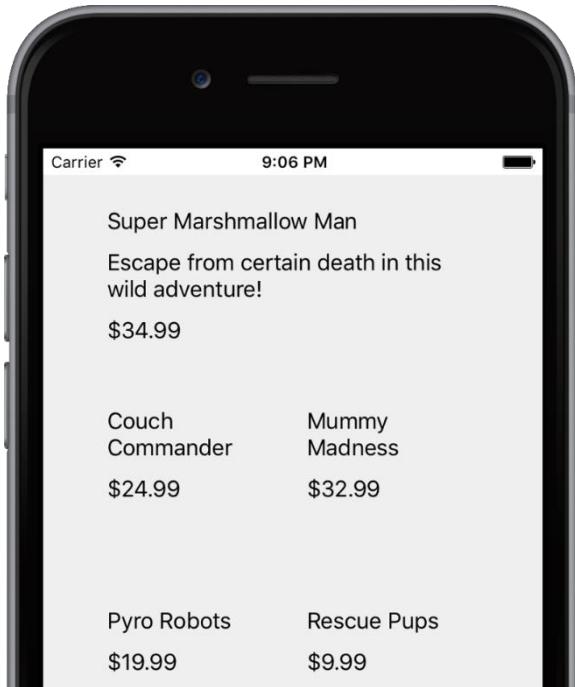


Figure 7.5 The Products page after removing all grid layout styling and setting the background color to a light grey.

ADDING SPACE AND A BACKGROUND COLOR

Our next task is to set the background color of the grid cells to white and add a margin between each cell. These changes don't necessarily apply to the entire app, so they should go into a page-specific CSS file. Create a new file named *product.css* and place it in the same directory as the *products.xml* file. With the page-specific CSS file created, create a *tile* class selector that will be used to represent each grid cell. Add a `background-color` and `margin` property to the *tile* class selector, as shown in listing 7.6.

Listing 7.6 Additions to the *products.css* file to make the grid cells stand out

```
.tile {  
    background-color: #FFFFFF;  
    margin: 2;  
}
```

Let's use the *tile* class we just created to apply these styles to each grid cell. Add `class="tile"` to each of the `StackLayout` elements on the Products page. This change makes a dramatic difference to the Tekmo app, as seen in figure 7.6.

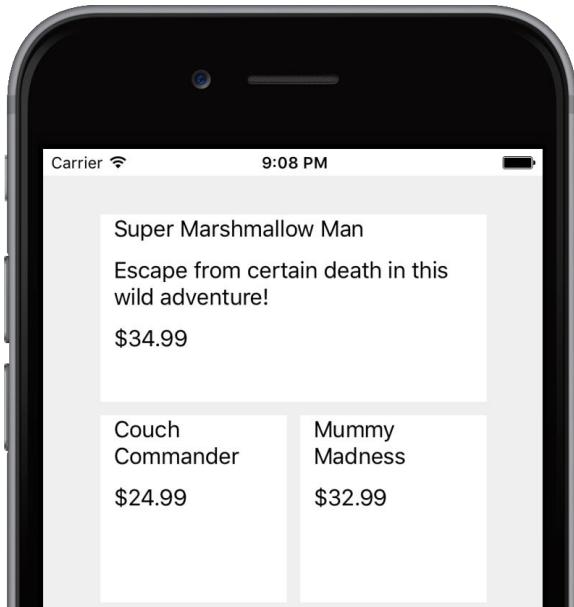


Figure 7.6 The Products page with grid cells identified with a white background color.

ADDING A TITLE BANNER

The title banner is a solid band or strip of color containing the game's title across the top of each grid cell. At first, this change seems straight forward: set the background color of the title label, but figure 7.7 shows what happens when we take this approach. It's not the result we want.

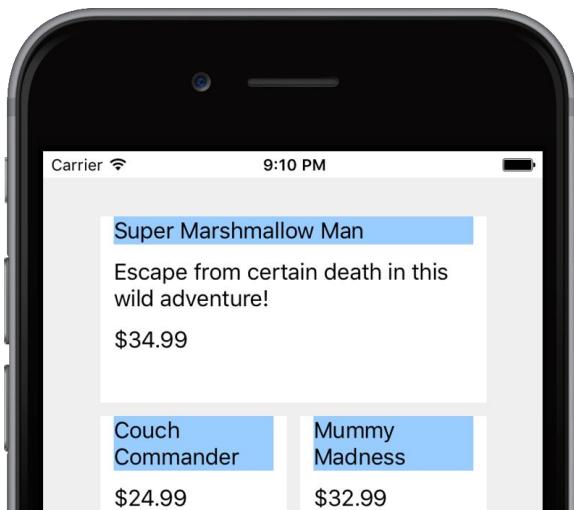


Figure 7.7 The Products page after with a background-color style applied to the tile labels.

When you set the background color of a label, NativeScript literally does just that: it sets the background of the label. Unfortunately, a label's background only falls underneath the inner text. The resulting effect of setting the background color is technically correct, but it doesn't look visually appealing. What we really want is a banner of color that stretches the entire width of each grid cell. Luckily, there's an easy way to do this.

TIP To add a solid banner or block of color to a page, add a stack layout and style the background color of the stack layout.

Using this tip, wrap a stack layout around the title label and apply a background color via a class selector. Listings 7.7 shows the changes to the products.css file and how to change one of the grid cells from the Products page by wrapping the title label in a stack layout targeting the `tile-title` CSS class.

Listing 7.7 Additions to the products page files to style the background color of the tile banner

```
.tile-title {                      #A
    background-color: #99ccff;  #A
}

<StackLayout row="0" col="0" colSpan="2" class="tile">          #B
    <StackLayout class="tile-title">                                #B
        <Label text="Super Marshmallow Man" textWrap="true" />      #B
    </StackLayout>                                         #B
    <Label textWrap="true" text="Escape from certain death in #B
        this wild adventure!" />                                #B
    <Label text="$34.99" />                                     #B
</StackLayout>                                         #B

#A Add to the products.css file
#B Update the products.xml file to add a tile-title class to the stack layout wrapping the title labels
```

After applying the same stack layout wrapping strategy to the remaining grid cells on the Products page, you will achieve the desired effect, as shown in figure 7.8.

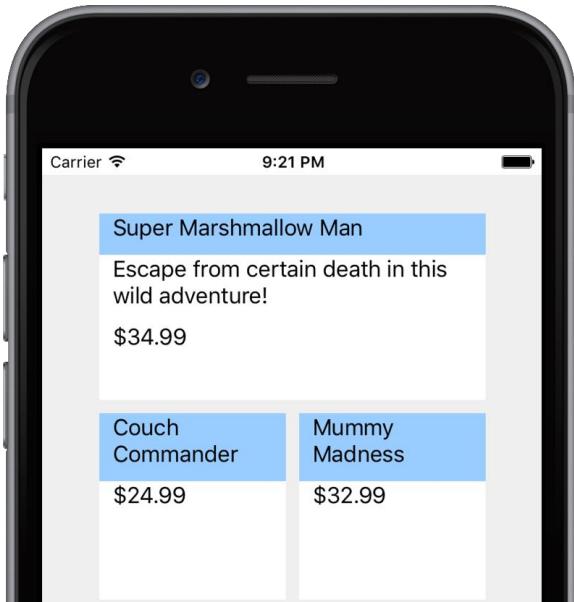


Figure 7.8 The Products page after correctly styling the grid cell titles via the background color of a stack layout wrapped around the title labels.

STYLING THE GRID CELL TEXT

The next change we'll make is to adjust the color and position of the title banner label and price label by adding style rules to the products.css file. Add a class attribute with a value of `price` to each price label, then add the CSS styles outlined in listing 7.8. Figure 7.8 shows the resulting UI changes.

Listing 7.8 Additions to the products.css file to style the background color of the tile banner

```
.tile-title Label {  
    font-size: 14;  
    color: black;  
    margin-top: 5;  
}  
.price {  
    color: #009933; #A  
    text-align: right;  
}  
#A #009933 is green
```

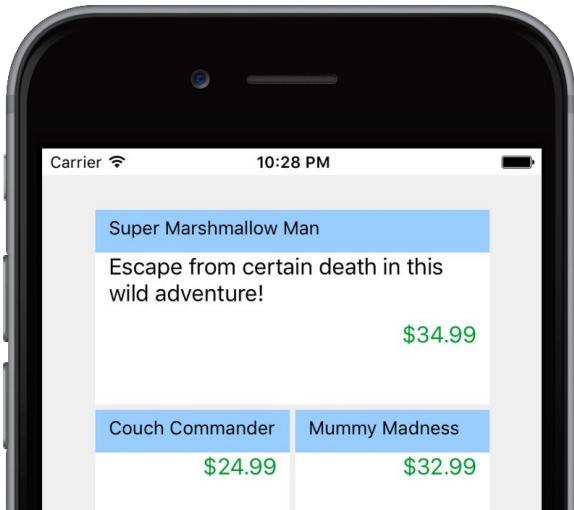


Figure 7.9 The Products page after applying styles to the titles and prices.

HIGHLIGHTING THE FEATURED PRODUCT

Now that we've added styles to each of the grid cells, let's turn our attention to the featured product: Super Marshmallow Man. It would be nice to make that grid cell stand out from the rest of the cells. One way to make this grid cell stand out is to apply an additional class to it and then use that additional class to *override* the styles already applied.

DEFINITION Overriding a CSS style is the process of defining a general style rule (like all labels with a class of `small` are font size 14) and selectively changing the value of the general style rule in certain circumstances. For example, if a label with the `small` class also has a class of `heading`, then the font size should be 16, rather than the default 14.

Let's add an additional class named `highlight` to the featured product grid cell, then override some of the style properties in the `products.css` file by adding rules for each of our tile-related classes when the tile also has the `highlight` class applied. Listing 7.9 outlines the CSS additions and listing 7.10 shows addition of the `highlight` class to the UI.

NOTE One way of *overriding* CSS styles is to apply an additional class to a parent element. The `highlight` class is in listing 7.9.

Listing 7.9 Additions to the `products.css` file to highlight the featured product

```
.highlight .tile-title {      #A  
    font-weight: bold;        #A  
    background-color: #6699ff; #A  
}  
#A
```

```
.highlight .tile-title Label {    #B
    font-size: 18;                #B
}
}

.highlight .price {               #C
    font-weight: bold;           #C
    color: red;                 #C
}
}

#A Change the title banner to have bold text and a slightly darker background color
#B Increase the title font size slightly
#C Make the price standout by making it bold and red
```

Listing 7.10 Additions to the products.xml file to highlight the featured product

```
<StackLayout row="0" col="0" colSpan="2" class="tile highlight">
    <StackLayout class="tile-title">
        <Label text="Super Marshmallow Man" textWrap="true" />
    </StackLayout>

    <Label textWrap="true" text="Escape from certain
        death in this wild adventure!" />
    <Label text="$34.99" class="price" />
</StackLayout>
```

In figure 7.10, you'll notice the results of these style changes: the title banner is slightly darker, the title text is larger and bold, and the price text is bold and a different color.

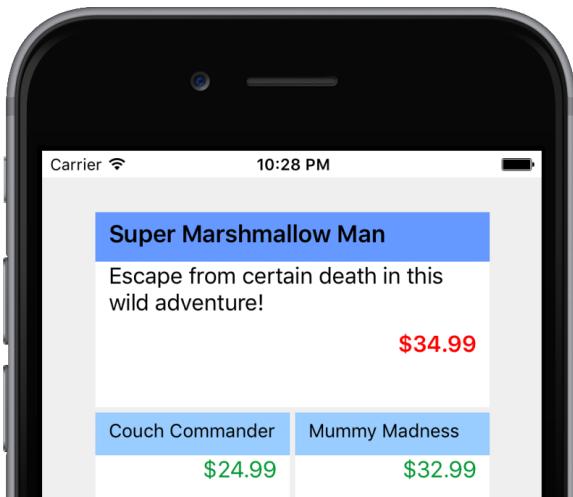


Figure 7.10 The Products page with the Super Marshmallow Man grid cell highlighted with a different style.

Great work! The CSS styles used to style the Tekmo app are by no means exhaustive of the capabilities in NativeScript, but they should serve as a launch point for you to feel empowered to try some ideas of your own.

7.2 Adding images to an app

In addition to styling text, adding background colors, and creatively arranging UI elements with borders and margins, images are another powerful tool in your tool belt for transforming “blah” apps into something beautiful. In this section, you’ll learn how to add images to an app by further refining the Tekmo app’s Products page.

7.2.1 Using the Image element

Let’s get right to it and add images to each of the grid cells on the Products page by adding an `<Image>` element to each grid cell.

DEFINITION An image is a JPEG or PNG graphic that will be displayed in the UI of an app. To add an image, you use the `<Image />` element.

Listing 7.11 contains the images to be added to the Products page. When adding the images, place them within the stack layout with the `tile` class, directly below the stack layout with the `tile-title` class.

NOTE You can find the images used in this section by downloading a zip file containing the images from

<https://github.com/mikebranstein/TheNativeScriptBook/blob/master/Chapter7/Tekmo/app/images/high-res-game-images.zip>.

Listing 7.11 Adding an image to each grid cell in the products.xml file

```
<Page xmlns="http://schemas.nativescript.org/tns.xsd">
  <GridLayout rows="*,*,*,*" columns="*,*" width="300" height="600">
    <StackLayout row="0" col="0" colSpan="2" class="tile highlight">
      <StackLayout class="tile-title"> #A
        <Label text="Super Marshmallow Man" textWrap="true" /> #A
      </StackLayout>
      <Image src "~/images/super-marshmallow-man.png" /> #B
      <Label text="$34.99" class="price" />
    </StackLayout>
    <StackLayout row="1" col="0" class="tile">
      <StackLayout class="tile-title">
        <Label text="Couch Commander" textWrap="true" />
      </StackLayout>
      <Image src "~/images/couch-commander.png" />
      <Label text="$24.99" class="price" />
    </StackLayout>
    <StackLayout row="1" col="1" class="tile">
      <StackLayout class="tile-title">
        <Label text="Mummy Madness" textWrap="true" />
      </StackLayout>
      <Image src "~/images/mummy-madness.png" />
      <Label text="$32.99" class="price" />
    </StackLayout>
    <StackLayout row="2" col="0" class="tile">
      <StackLayout class="tile-title">
```

```
<Label text="Pyro Robots" textWrap="true" />
</StackLayout>
<Image src "~/images/pyro-robots.png" />
<Label text="$19.99" class="price" />
</StackLayout>
<StackLayout row="2" col="1" class="tile">
    <StackLayout class="tile-title">
        <Label text="Rescue Pups" textWrap="true" />
    </StackLayout>
    <Image src "~/images/rescue-pups.png" />
    <Label text="$9.99" class="price" />
</StackLayout>
<StackLayout row="3" col="0" class="tile">
    <StackLayout class="tile-title">
        <Label text="Vampire Valkyrie" textWrap="true" />
    </StackLayout>
    <Image src "~/images/vampire-valkyrie.png" />
    <Label text="$21.99" class="price" />
</StackLayout>
</GridLayout>
</Page>
```

**#A Place the image directly below the stack layout with a class attribute of tile-title
#B The game images from above, placed below the stack layout**

After adding the images to the Products page, it's starting to come together, as shown in figure 7.11.

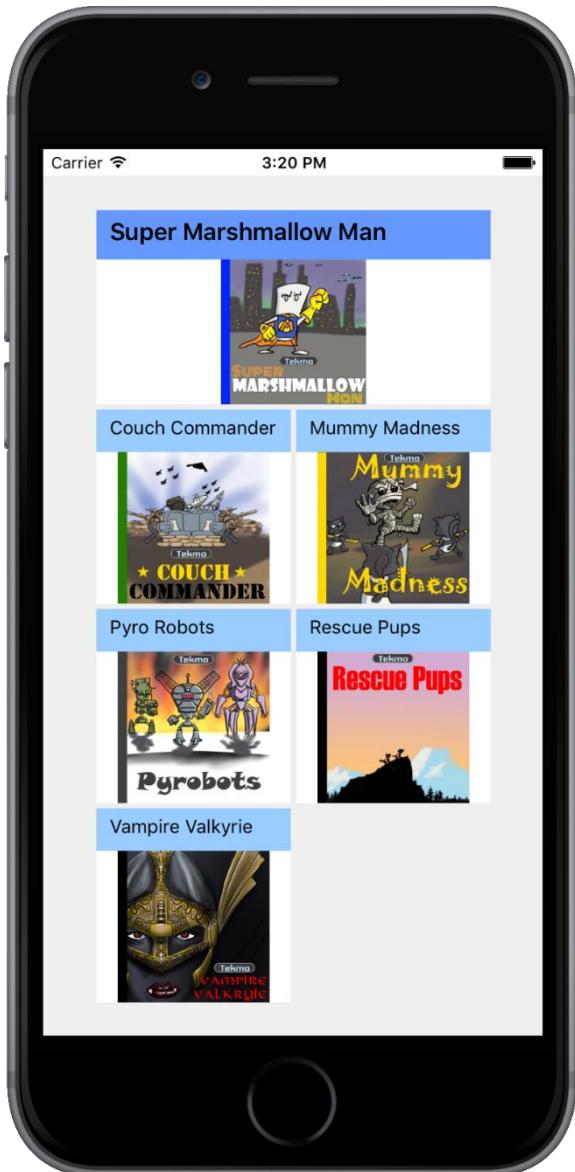


Figure 7.11 The Products page after images have been added to the XML code.

I want to call your attention to the markup of the `image` element. NativeScript images are like HTML images, specifically in how they reference image files to load using the `src` attribute. You can load both local and remote image files by using a file name or a file in the same folder (for example, `image-`

`name.png`), a relative file path and file name of an image in a different folder (`..\..\image-name.png`), or a URL. The syntax is similar to how images are loaded in HTML, and summarized in table 7.2 as a point of reference.

Table 7.2 Different methods for loading image resources

Method	Syntax	Description
relative file path	<code>src="{image-file-path}"</code>	Loads an image from a location within the app, relative to the folder the page is located in
URL	<code>src="http://image-url"</code> <code>src="https://image-url"</code>	Loads an image from a URL using HTTP or HTTPS
resource	<code>src="res://image-name"</code>	Similar to relative file path, but loads the image from the App_Resources folder. No file name extension is needed.

One difference from the HTML image syntax you'll notice is the *resource* method for loading images.

DEFINITION Loading images by resource is a way of loading different versions of an image based upon a device's resolution.

It's important that you know about this method, because it is an extremely powerful capability of NativeScript (and makes your life easier when developing cross-platform apps).

Before you learn about the specifics of resource loading, however, you'll need a little more background on mobile devices. In chapter 3, you learned about the varying screen resolutions and DPI. You'll need to draw on this past knowledge in the next section, so let's review what DPI means before we jump in.

DEFINITION Dots per inch (DPI) is a measure of dot density, and is usually used in the printing industry to describe the number of printed dots appearing in a square inch of a printed book or magazine. When referring to screens, the concept of a "dot" is often confused with a "pixel." Screens have pixels, not dots; therefore, their density is measured in pixels with pixels per inch (PPI). Although DPI and PPI are technically different, most people don't differentiate between the two. In fact, the Android platform prefers the terminology of DPI versus PPI. Through this book, I will use the term DPI.

7.2.2 Challenges with displaying images on mobile devices

It's important to understand (at a high-level) the challenges in consistently displaying images on various mobile devices.

WARNING This section may scare you initially, and that's because cross-platform device DPIs and the various requirements for Android and iOS can be confusing. But don't worry: NativeScript does a good job of abstracting away the complexities of cross-platform images. Stick with us!

One of the core problems in displaying images consistently is the sheer number of mobile devices, each with varying screen size, resolution, and DPI. Let's use the iOS device ecosystem as an example. iOS devices are part of a highly-controlled hardware ecosystem, resulting in fewer variances across the models. Now, consider the different iPhone models as of mid-2016: 2G, 3G, 3Gs, 4, 4s, 5, 5s, 5c, 6, 6 plus, 6s, 6s plus, 7, and SE. There are 13 different models at the time of this book, with anywhere between four and seven various screen resolutions (depending how you count), and displayed at three different DPI densities.

NOTE If you're interested in learning more about the various iPhone screen resolutions and DPIs, check out <https://www.paintcodeapp.com/news/ultimate-guide-to-iphone-resolutions>. This guide has an excellent visual explaining how iOS maps and transforms images to the different screen sizes and DPI settings.

Yow! That's only iOS, the most standardized hardware device ecosystem. Android has similar differences, but it's across an even larger hardware space, with five different DPI densities. I don't want to sound all doom-and-gloom to you, so let me back-pedal. There are well-defined standards and guidelines for displaying images on both Android and iOS platforms, so my point isn't to scare you away, just illustrate some of the complexity and challenges.

THE EFFECT OF VARIABLE SCREEN DPI

So, what does this mean for you? Unfortunately, a lot. But, I'm going to take it slow. As a mobile developer, it's critical to understand the differences between platforms, because it's an important aspect of cross-platform mobile development.

A core challenge in developing images for multiple DPI densities is satisfying the expectations of your users. Older mobile devices tend to have lower-DPI displays. You can design and create beautiful high-resolution images, but the screen just isn't capable of displaying the image so it looks crisp and clear. Take the Super Marshmallow Man image as an example (figure 7.12), displayed on a low-DPI screen (163 DPI) and high-DPI screen (401 DPI).



Figure 7.12 Side-by-side images of a low DPI display (163 DPI, on left) and a high DPI screen (401 DPI, on right) displaying the same high-resolution image. Note the low DPI screen appears to have a blurry image.

The difference between the images is subtle, but if you look closely at the text, it's easy to see. Figure 7.13 shows the images zoomed in for a closer view.



Figure 7.13 A zoomed view of a low- (left) and high- (right) DPI display showing the same image.

From this closer look, you can see the low-DPI display on the left is much blurrier than the image on the right. Having an older and low-DPI phone, I expect everything on my phone to appear blurry, but if I had a high-DPI phone, I'd want my images to be crisp and clear all the time.

So, one of the core problems in developing images across various DPIs is ensuring your images look as good as possible on every device.

7.2.3 Solutions to DPI density differences

Let's recap the problem at hand. Between Android and iOS, there are a total of nine different screen DPIs, and when we display an image across each of these devices, we want the image to be crisp and clear (or as crisp and clear as they can be based upon the device's DPI).

There are a lot of ways to solve this problem, but I'm going to focus on two: a simple way and a brute-force way. Let's tackle the simple way first, and discuss why you may not want to use this solution. Then, we'll talk about a solution I call brute force because it requires a lot of work but will give your users a better experience.

THE SIMPLE SOLUTION

The simple solution is to use high-resolution images in your app always. This guarantees that users with lower DPI phones get the best possible experience for every device. When you use this method, you want to also specify a display size for the image. Let's continue to use the Super Marshmallow Man image as an example, stretching it to fill the screen. Listing 7.12 contains the code used to display the image.

Listing 7.12 Displaying an image to fill the entire screen across any device

```
<Page>
  <GridLayout rows="*" columns="*"> #A
    <Image src "~/images/super-marshmallow-man.png" />
  </GridLayout>
</Page>
#A A grid layout with 1 row and 1 column set to * will expand to fill an entire screen
```

By creating a grid layout with one row and column set to the default (*) sizing option will expand the grid's content to fill the entire page. The image inside the grid will expand to the full size of the screen.

Figure 7.14 shows the image displayed on an iPhone 3GS (163 DPI) and 6 Plus (401 DPI). Note that I've zoomed in on the 3GS so it's physical size appears the same as the 6 Plus (just imagine you're holding it closer to your face). The screen resolution of the iPhone 3G is substantially smaller than the iPhone 6 Plus, so the image will appear blurry (but it *is* still the best possible result on both devices).



Figure 7.14 High-resolution images displayed on the iPhone 3G (left) and iPhone 6 Plus (right). Note the blurriness, especially on the iPhone 3G.

This approach is simple and it gives the best possible user experience (visually), but you're loading a high-resolution image on low-resolution displays. That's inefficient, and could lead to performance problems. You really want to load images that are "high enough" resolution to look right on each device.

WARNING Don't load high-resolution images on low DPI devices. Devices with low-resolution displays typically have less processing power and memory, therefore, optimizing your app as much as possible will lead to overall better behavior. Now, this is a generalization that may not apply in every circumstance, but better safe than sorry.

There is a better way.

SOLVING THE MULTIPLE-DPI DEVICE PROBLEM

Because the simple solution could result in poor user experience, let's discuss a better way. The answer to this problem is straightforward, but rather brute force.

SOLUTION If you have nine varying DPIs, create nine different versions of each image (one for each varying DPI).

See, I said it was brute-force. Because this solution is so labor intensive, we're going to explain how to do it at a high-level, but then give you a way to cheat and skip the manual steps:

- Determine the highest DPI you'll be supporting (for example, 401 DPI for iPhone 6 Plus).
- Decide the maximum visual size of the displayed image (for example, 1 inch).
- Calculate the minimum dimensions for your image (for example, for a square image, 1 inch x 401 DPI = 401 x 401 pixels).
- Create a crisp and clear image with the minimum dimensions, saving this as an original base image.
- Calculate the varying smaller image dimensions you need based on the varying device DPIs (for example, for iOS you'll also need 326 DPI and 163 DPI, resulting in 2 more images 326 x 326 pixels and 163 x 163 pixels).
- Using the base image, shrink the base image down to the other sizes calculated in the previous step, making sure you start with the saved base image each time you shrink to another DPI.

What a pain! You'll have to do this up to nine times to support all of the various DPIs for both Android and iOS. But, no fear, we've got you covered. We built a website dedicated to resizing your images for Android and iOS: <http://nsimage.brosteins.com>. To use the site, upload a single high-resolution image. After uploading, the image is resized to all the various image resolutions needed for Android and iOS. A zip file containing all the variations is then downloaded.

TIP Don't manually shrink your images, use an online service like the Brostein's image creator, <http://nsimage.brosteins.com>.

If you stick with this solution, you'll be guaranteed to have crisp and clear images that display in your apps.

7.2.4 *Displaying multi-resolution images in NativeScript apps*

Now that you learned the process for making images crisp and clear, you've got up to nine different images! How can you display these, across the various devices, and know which image to use in which circumstance? Don't worry: NativeScript makes this easy with something called image resources.

DEFINITION Image resources make it easy to load the right image for each platform. Like other aspects of NativeScript, image resources rely on conventions. If you name your collection of images with a specific file-naming convention and place them in the App_Resources folder of your app, NativeScript will automatically load the correct image.

I'll explain how and where to place image resources in a few minutes, but let's cover the easy part first: referencing an image resource in the image element. To add a reference to an image resource, change the `src="{image-file-name}"` attribute of the image element to `src="res://{{image-file-name-without-extension}}"`. Listing 7.13 shows the changes you'll have to make to the Products page to switch each image.

Listing 7.13 Using image resources to load images

```
<Image src="res://super-marshmallow-man" /> #A
<Image src="res://couch-commander" /> #A
<Image src="res://mummy-madness" /> #A
<Image src="res://pyro-robots" /> #A
<Image src="res://rescuss-pups" /> #A
<Image src="res://vampire-valkyries" /> #A
```

#A Replace the existing image elements with the image resource version

When you add an image element that references an image resource, you don't need to specify the file extension, just the file name preceded by `res://`, as shown in listing 7.12.

Now that you've learned how to reference image resources, let's tackle the more laborious part: creating the multiple image versions for Android and iOS. In chapter 3, you learned that the `App_Resources` folder holds platform-specific files. Android customizations are located inside the `Android` folder, and iOS customizations in the `iOS` folder. Unfortunately, each platform organizes their platform-specific files a bit differently, so we'll look at each platform separately, beginning with Android.

ANDROID IMAGE RESOURCES

Let's take a moment and explore the Android-specific contents of the `App_Resources` folder (figure 7.15).

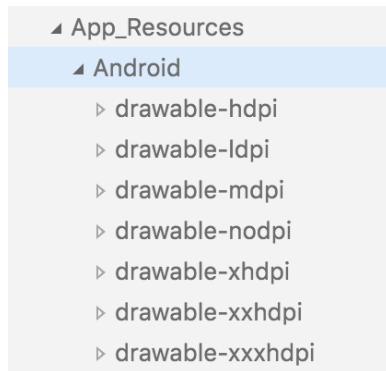


Figure 7.15 The `App_Resources` Android folder showing the DPI density folder structure.

Inside the `Android` folder, note the six folders named `drawable-{size}dpi`. These folders correspond to the six different device DPI densities on the Android platform.

NOTE Due to the large ecosystem of Android devices, the six DPI densities aren't exact. Instead, approximate DPIs are used, because they can vary between similar devices.

Table 7.3 summarizes the approximate DPIs for each of the six categories.

Table 7.3 Approximate Android device DPI densities and the corresponding `App_Resources` folder

Size	App_Resources Folder	Approximate DPI
------	----------------------	-----------------

low	drawable-ldpi	~120 DPI
medium	drawable-mdpi	~160
high	drawable-hdpi	~240
extra-high	drawable-xhdpi	~320
extra-extra-high	drawable-xxhdpi	~480
extra-extra-extra-high	drawable-xxxhdpi	~640

To use image resources on Android, you place an appropriate-size image in each of the *drawable-{size}dpi* folders. The files should all be named identically (see figure 7.15).

Let's use the Super Marshmallow Man image from earlier in the chapter and create an image for each Android DPI density.

- Start with a high-DPI version of the super-marshmallow-man.png file. You can download it from:
 - <https://github.com/mikebranstein/TheNativeScriptBook/blob/master/Chapter7/Tekmo/app/images/high-res-game-images.zip>
- Use our website <http://nsimage.brostains.com> to create the various device-specific resolutions for each of the high-resolution images in the zip file. If you're a designer and would rather manually convert the images manually, grab your favorite image editor and have at it (you can reference table 7.3 to get the right DPI for each image)
- Save each image to the corresponding folder in the *App_Resources/Android* folder of your NativeScript app. Be sure the names are all identical (for example, super-marshmallow-man.png)

After you've finished creating the six images from the original *super-marshmallow-man.png* file, you should have a file named *super-marshmallow-man.png* in each of the folders (figure 7.16).

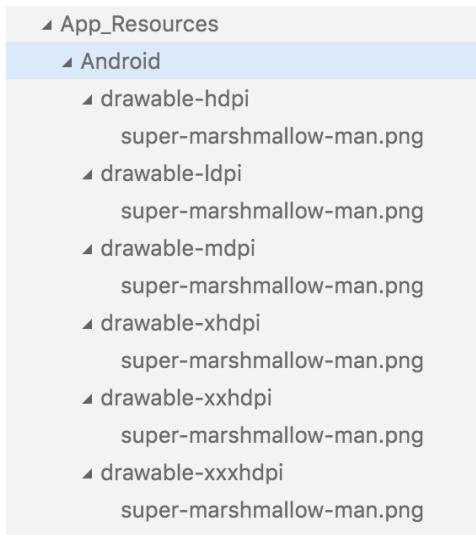


Figure 7.16 Image resources are placed in each drawable folder. Files are named the same.

If you were to open several of the images side-by-side, you'll notice the varying sizes. Figure 7.17 shows the low-, high-, and extra-extra-high-resolution versions. As you'd expect the extra-extra-high version appears approximately four times the size as the low-resolution version (~ 480 dpi / ~ 120 dpi = $\sim 4x$).



Figure 7.17 ldpi, hddpi, and xxhdpi images compared side-by-side to visual the difference in size (shown from left to right).

iOS IMAGE RESOURCES

The App_Resources/iOS folder is organized differently, compared with the Android folder. Figure 7.18 shows the iOS folder from the Tekmo app.

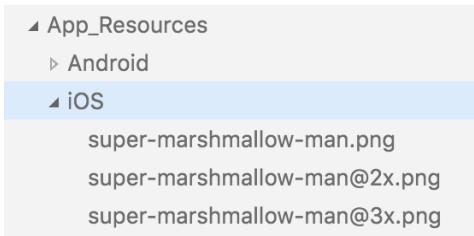


Figure 7.18 iOS image resources for the Super Marshmallow Man image. Note the lack of nested folders and the file naming differences.

First, you'll notice there's no folder hierarchy related to images and device DPI densities. Instead, all DPI-specific images are placed in the root of the iOS folder.

Second, image resources for iOS are named differently, and correspond to the various DPI densities on iOS. Like Android's ldpi, mdpi, ..., xxxhdpi convention, iOS has three different device DPI densities that it supports (163, 326, and 401 DPI). These densities correspond to a specific file-naming convention. Table 7.4 details each DPI density and associated file name.

Table 7.4 iOS device DPI densities and the corresponding App_Resources file name

Size	File name	DPI
1x	{file-name}.{extension}	163 DPI
2x	{file-name}@2x.{extension}	326
3x	{file-name}@3x.{extension}	401

Let's create iOS-specific images with the same Super Marshmallow Man image we used previously.

- Start with the same high-DPI version of the super-marshmallow-man.png file downloaded from:
 - <https://github.com/mikebranstein/TheNativeScriptBook/blob/master/Chapter7/Tekmo/app/images/high-res-game-images.zip>
- Create three images, each with a corresponding target DPI, as shown in table 7.4. Again, you're welcome to use an image-editing program of your choice, but we prefer to use the image re-sizer at <http://nsimage.brosteins.com> to automate the process.
- Save the three images to the `App_Resources/iOS` folder of your NativeScript app. The images should be named: `super-marshmallow-man.png`, `super-marshmallow-man@2x.png`, and `super-marshmallow-man@3x.png`.

Now that we have generated image resources for Android and iOS, named the files properly, added them to the correct App_Resources folder, and changed the image element to use the `res://` syntax, we're ready to check out the results. Figure 7.19 shows the Products page after adding the images.

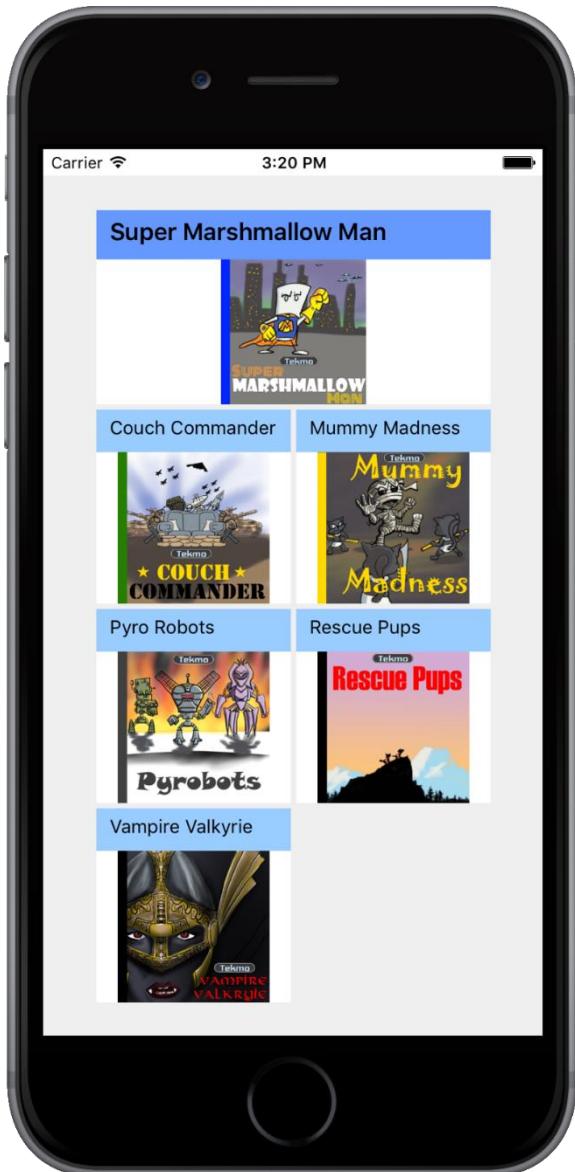


Figure 7.19 The Products page with image resources added.

As you can see, it's not quite right, because the images fill the grid cells, pushing out the prices, so we'll need to add some styling. If you haven't been following along, I've included the XML code for the

Products page in listing 7.14. Use this code as a reference so we can finish styling the Products page together.

Listing 7.14 Complete Products page code with image resources added

```
<GridLayout rows="*,*,*,*" columns="*,*" width="300" height="600">
  <StackLayout row="0" col="0" colSpan="2" class="tile highlight">
    <StackLayout class="tile-title">
      <Label text="Super Marshmallow Man" textWrap="true" />
    </StackLayout>
    <Image src="res://super-marshmallow-man" />
    <Label textWrap="true" text="Escape from certain death
      in this wild adventure!" />
    <Label text="$34.99" class="price" />
  </StackLayout>
  <StackLayout row="1" col="0" class="tile">
    <StackLayout class="tile-title">
      <Label text="Couch Commander" textWrap="true" />
    </StackLayout>
    <Image src="res://couch-commander" />
    <Label text="$24.99" class="price" />
  </StackLayout>
  <StackLayout row="1" col="1" class="tile">
    <StackLayout class="tile-title">
      <Label text="Mummy Madness" textWrap="true" />
    </StackLayout>
    <Image src="res://mummy-madness" />
    <Label text="$32.99" class="price" />
  </StackLayout>
  <StackLayout row="2" col="0" class="tile">
    <StackLayout class="tile-title">
      <Label text="Pyro Robots" textWrap="true" />
    </StackLayout>
    <Image src="res://pyro-robots" />
    <Label text="$19.99" class="price" />
  </StackLayout>
  <StackLayout row="2" col="1" class="tile">
    <StackLayout class="tile-title">
      <Label text="Rescue Pups" textWrap="true" />
    </StackLayout>
    <Image src="res://rescue-pups" />
    <Label text="$9.99" class="price" />
  </StackLayout>
  <StackLayout row="3" col="0" class="tile">
    <StackLayout class="tile-title">
      <Label text="Vampire Valkyrie" textWrap="true" />
    </StackLayout>
    <Image src="res://vampire-valkyrie" />
    <Label text="$21.99" class="price" />
  </StackLayout>
</GridLayout>
```

7.2.5 Styling images

Now that we've added our images to the Tekmo app, we have a little bit of clean up to do. The first thing we should do is shrink the images to a reasonable size (~80 pixels). Add an `Image { width: 80; height:80; }` style to the `products.css` file. The results are shown in figure 7.20.

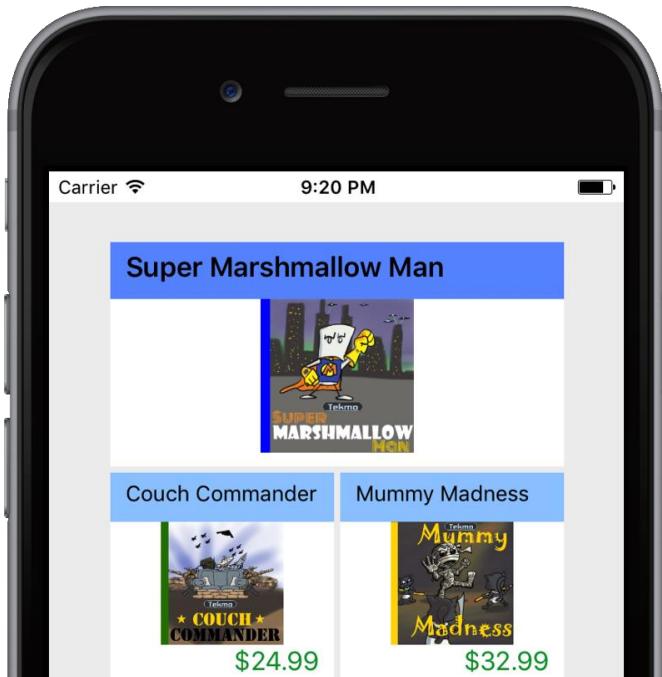


Figure 7.20 The Products page after styling the images to reduce their size and including the price of the game.

Changing the image size looks OK, but the Super Marshmallow Man text and price has fallen out of the grid cell. The highlighted image should be left-aligned, with the description and price aligned to the right of the image.

There are a variety of ways to accomplish this style. Two ways are to do the following:

- Nest a 1-row, 2-column grid layout inside of the stack panel.
- Use a series of nested stack layouts, one stacking UI elements horizontally, and another stacking the UI vertically.

You may think of a third or fourth way to organizing your UI, but let's use the stack layout approach because it'll help you learn about another aspect of the stack layout: orientation.

DEFINITION The orientation property of a stack layout tells NativeScript whether to render the layout's contents vertically or horizontally. By default, content is rendered vertically, but you can change it to render horizontally by adding the `orientation="horizontal"` attribute.

Add several stack layouts to the Super Marshmallow Man XML code, as shown in listing 7.15.

Listing 7.15 Products page code with image resources added

```
<StackLayout row="0" col="0" colSpan="2" class="tile highlight">
    <StackLayout class="tile-title">
        <Label text="Super Marshmallow Man" textWrap="true" />
    </StackLayout>
    <StackLayout orientation="horizontal"> #A
        <Image src="res://super-marshmallow-man" />
    <StackLayout> #B
        <Label textWrap="true" text="Escape from certain death #B
            in this wild adventure!" /> #B
        <Label text="$34.99" class="price" /> #B
    </StackLayout> #B
</StackLayout>
</StackLayout>
#A Wrap a horizontal stack layout around the image and labels
#B A second vertical stack layout will wrap the labels
```

Now, let's increase the image size to 100 x 100 pixels by adding changing the image element style for the highlighted image to `.highlight Image { width: 100; height: 100; }`. With these changes, we have the final version of the Products page (figure 7.21).

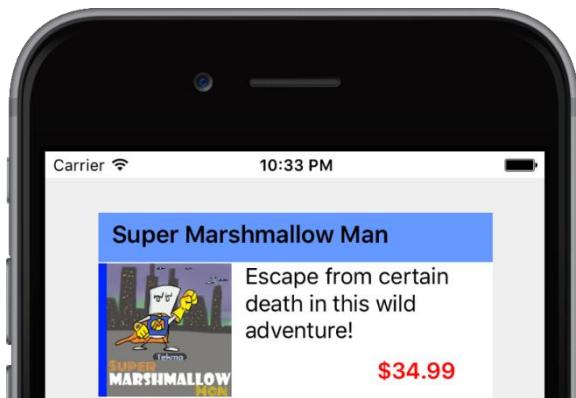


Figure 7.21 The Products page after styling Super Marshmallow Man to include nested stack layouts with a blend of horizontal and vertical orientations.

A final version of the code for chapter 7 can be found at <https://github.com/mikebranstein/TheNativeScriptBook/tree/master/Chapter7>.

7.3 Summary

In this chapter, you learned that:

- A subset of CSS style properties can be used to style the UI of NativeScript apps
- Style can be set globally (in the app.css file), on a page-by-page basis (using the page-name.css file), and inline with the XML code (with a style="..." attribute)
- Images can be added to NativeScript apps in three ways: locally (via a relative path and file name), online (via http or https), and through a resource reference (using res:// and image resources in the App_Resources folder)
- When loading images from the App_Resources folder, up to nine different images are needed to support all device DPI densities on Android and iOS (Android requires six, iOS requires three)
- You can manually create App_Resources images using your preferred image editing tool or the automated image resizer at <http://nsimage.brostiens.com>.

7.4 Exercises

In this chapter, you learned how to style NativeScript apps with CSS. Use what you've learned in this chapter and previous chapters to do the following:

- Create a page that displays a green background when in portrait mode, but red when in landscape.
- Using figure 7.22 as a point of reference, design a page to mimic the page's content and styling as closely as possible. Hint: use the border-radius CSS property to get rounded corners.

Height	Category
2' 04"	Seed
Weight	
15.2 lbs	

Figure 7.22 An image with multiple rows and columns, varying fonts and colors, and a background.

7.5 Solutions

To create a page that displays a green background when in portrait mode, but red when in landscape do the following:

- Assume the page name is main-page.xml.
- Create two page-specific XML files named main-page.port.css and main-page.land.css.
- Add `Page { background-color: green; }` to the main-page.port.css file.

- Add `Page {background-color: red; }` to the `main-page.land.css` file.

To mimic the page's content displayed in figure 7.22, do the following:

- Add the markup in listing 7.16 to the page.

Listing 7.16 XML markup to create the rounded-edge information box

```
<GridLayout rows="auto,auto,auto,auto" columns="*,*"  
    class="info-container">  
    <Label text="Height" row="0" col="0" class="info info-title" />  
    <Label text="2' 04"" row="1" col="0" class="info info-value" />  
  
    <Label text="Category" row="0" col="1" class="info info-title" />  
    <Label text="Seed" row="1" col="1" class="info info-value" />  
  
    <Label text="Weight" row="2" col="0" class="info info-title" />  
    <Label text="15.2 lbs" row="3" col="0" class="info info-value" />  
</GridLayout>
```

- Add the CSS styles in listing 7.17 to the page's CSS file.

Listing 7.17 CSS styles to create the rounded-edge information box

```
.info-container {  
    margin-bottom: 20;  
    background-color: #30a7d7;  
    border-radius: 20;  
    padding: 20;  
    padding-bottom: 5;  
}  
  
.info-title {  
    margin-bottom: 10;  
    color: white;  
    font-size: 20;  
}  
  
.info-value {  
    margin-bottom: 15;  
    color: #212121;  
    font-size: 24;  
}
```

Part 3: Refining Your App

8

Working with data

This chapter covers

- How to use observable objects and observable arrays
- How to automatically update UI elements when observables change their value
- How a device's local storage can be used to save data

In the last few chapters, you took a deeper dive into creating apps with NativeScript through the lens of the Tekmo app. You learned how to design multi-page apps, navigate between pages, and how to organize and style your UI using layouts and CSS. In the Tekmo app, a lot of time was spent duplicating UI elements: take the products page as an example (figure 8.1).

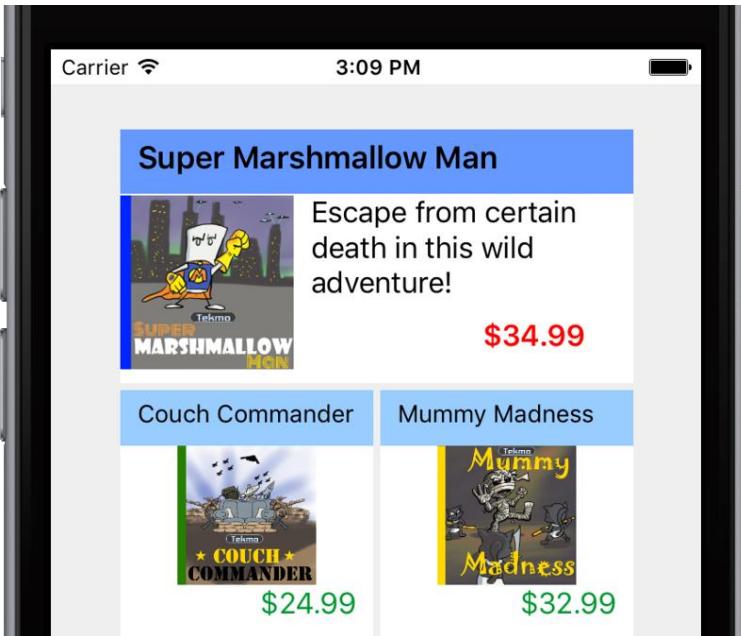


Figure 8.1 The products page from the Tekmo app, showing several hard-coded products.

Each of the products was hard-coded in the product page's XML file. Yeah, it's only six products, but what if there were dozens of products. Better yet, what if the products changed on a regular basis? In fact, updating hard-coded UI elements would become cumbersome rather quickly.

Let's take a different approach: imagine that we pulled the product listings for the Tekmo app from a file, database, or publically-accessible API endpoint? After retrieving this data, we could update the product listing with the retrieved data. This would allow us to write less code and our app would be more dynamic because we're no longer hard-coding everything.

In this chapter, we'll teach you how to write less code and make your apps more dynamic. At the same time, we'll be building a new app: the Pet Scrapbook. With the Pet Scrapbook, you'll be able to capture the fun moments of your pet's life by creating virtual scrapbook pages filled with images and captions. We'll go through several iterations of the app over the next several chapters, changing the app's code to use new aspects of NativeScript as you learn them.

Before we get started with the Pet Scrapbook, there are some new concepts you'll need to learn so we can abandon hard-coding everything and write less code. So, how do we actually write less code, while still allowing users to create multiple pages with their scrapbook? It may not be obvious, but we'll do this through the use of templates.

DEFINITION Templates are a way to create the UI element structure of page, while not adding the actual text or image data displayed. Once created, the UI elements in a template act as placeholders for the real element displayed on the screen.

In the Pet Scrapbook, we'll create a template representing the structure of a single scrapbook page. The template will contain placeholders for a pet's name, age, the title of the page, images, and captions. You may still be wondering how this will save you time, and allow you to write less code. Let's think about this in the context of a real scrapbook. Imagine you're creating a physical scrapbook, and want to add a page to the book. You get a piece of colored paper and start by organizing the page: measuring and using a ruler to ensure items are aligned, straight, and in a location similar to other pages in your scrapbook. You do this for each picture, sticker, and text you add to the page. This sounds pretty time-consuming. But what if you started with a template: a page that was already laid out with placeholders for the page's title, your pet's name, pictures, and other design elements? All of a sudden, adding a new page becomes much easier because the heavy lifting has been done and you need to worry about only the page's content.

Using templates in our app will be just like using a template in a real scrapbook. When you add a new page to the NativeScript pet scrapbook, you'll be able to use the same template, but display different details. Because we're reusing the same template, we'll also reuse the code we write for the page.

8.1 Databinding

Now that you know about templates, we want to introduce you to how a single UI template can be reused to display different details. The underlying technology used to do this is called *databinding*.

DEFINITION Databinding is the process of linking UI elements to objects in code. When a change is made to a UI element that is linked to an object in code, the change is reflected in the object or property. UI elements that are linked to objects in code are referred to as being *data-bound*.

Databinding is just the name for the overall process of linking together a JavaScript object and UI elements. Databinding is important because this is how we solve the problem of needing to hardcode products into the product page of the Tekmo app or updating an age field on the Pet Scrapbook app based on the birthday that a user enters. Before we dive further into using databinding, let's learn about one more concept that drives the inner-workings of databinding: *observables*.

DEFINITION Observables are special JavaScript objects that provide your code with notifications when one of their values changes.

We like to think of observables like kids in a classroom: every time something changes, they raise their hands to tell their teacher. It could be a runny nose, they need to use the bathroom, or just want to show their teacher the cool robot picture they just drew, it really doesn't matter what changed, but they'll raise their hand whenever something changes just to make sure their teacher knows about it. Kids are just like observable objects (also known as observables), except observables don't raise their hands, they raise an event. Figure 8.2 shows how an observable object raises an event when one of its internal values changes.

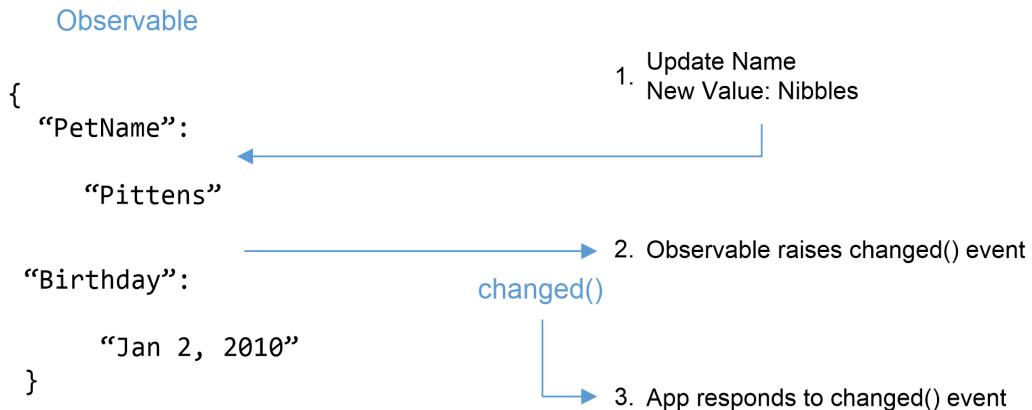


Figure 8.2 Your app can respond to an observable object when one of its internal values changes.

At this point, you may be wondering how all of this ties together. Templates, databinding, observables, events (oh my)! Together these concepts form the foundation that we will be working with to solve the hard-coding problem. Figure 8.3 shows the relationship between these 4 concepts.

Data Binding

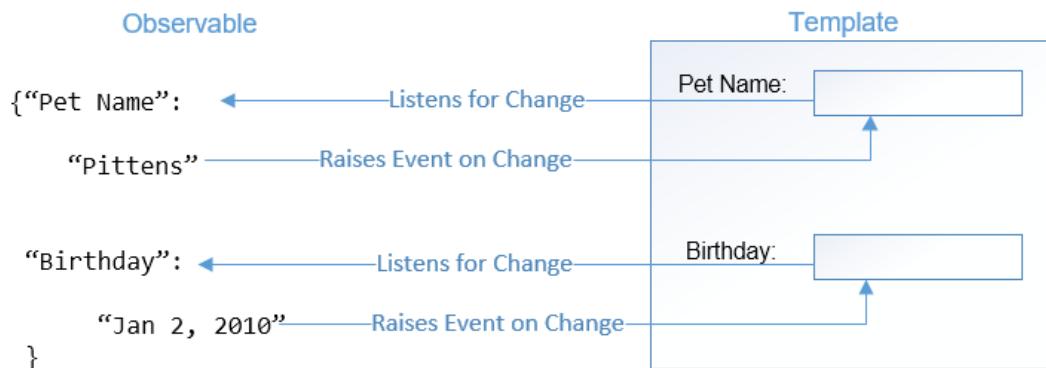


Figure 8.3 The relationship between databinding, templates, observables, and observable change events.

Databinding describes the act of linking together a UI template and an observable. Once linked, the template listens for change events to be raised by the observable. When a change happens to an observable's value, an event is raised. The registered event listener then responds to the observable's change by displaying the observable's new value.

Now that you have a general understanding of databinding, observables, templates, and observable events, let's take a look and see how we actually use these in code.

8.2 Observables in action

Go ahead and start a new blank project for the Pet Scrapbook app. Remember you can use the NativeScript CLI to scaffold out a new project for you by using the `tns create` command.

```
tns create PetScrapbook --template tns-template-blank
```

Similar to the Tekmo app that you previously created, add a `views` folder as shown in figure 8.4 to your project. This is where you will be adding views for the Pet Scrapbook app. The first page that we will be creating is the Home page, so add the `home-page.xml`, `home-page.js`, and `home-page.css` files to the `views` folder.

NOTE Don't forget to change the page in the `app.js` file that is loaded when the app loads to the `home-page.xml` view that we added.

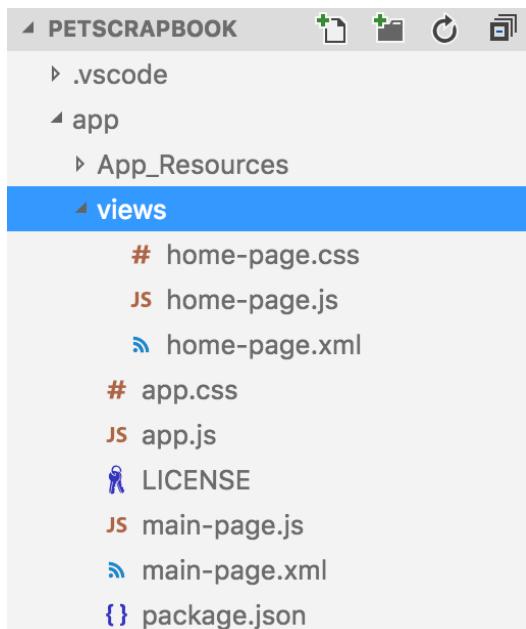


Figure 8.4 The resulting folder structure of the Pet Scrapbook app after scaffolding a blank NativeScript app and creating a `views` folder.

Let's start off by creating an observable, setting several values, and seeing how it raises an event when one of the values changes. Add the code from listing 8.1 to the home-page.js file.

Listing 8.1 The views/home-page.js file creating and observable, setting values, and listening for changes

```
var observableModule = require("data/observable");
var pet = new observableModule.Observable();

pet.set("Name", "Riven");
pet.set("Type", "Dog");
pet.set("Age", 3);

pet.on ("propertyChange", function(eventData){ // #A
    var changedPet = eventData.object; // #B

    console.log("Your pet is a " + changedPet.Type + " named "
        + changedPet.Name + " and is " + changedPet.get("Age") + " years old.");
});

pet.set("Age", 4); // #C
#A Creating an event handler
#B Getting the object that changed
#C Updating the value causes the property change event to occur
```

We think it's helpful to visualize what's going on in listing 8.1, so reference figure 8.5, which shows a sequence diagram of the major events of the observable's property changed process.

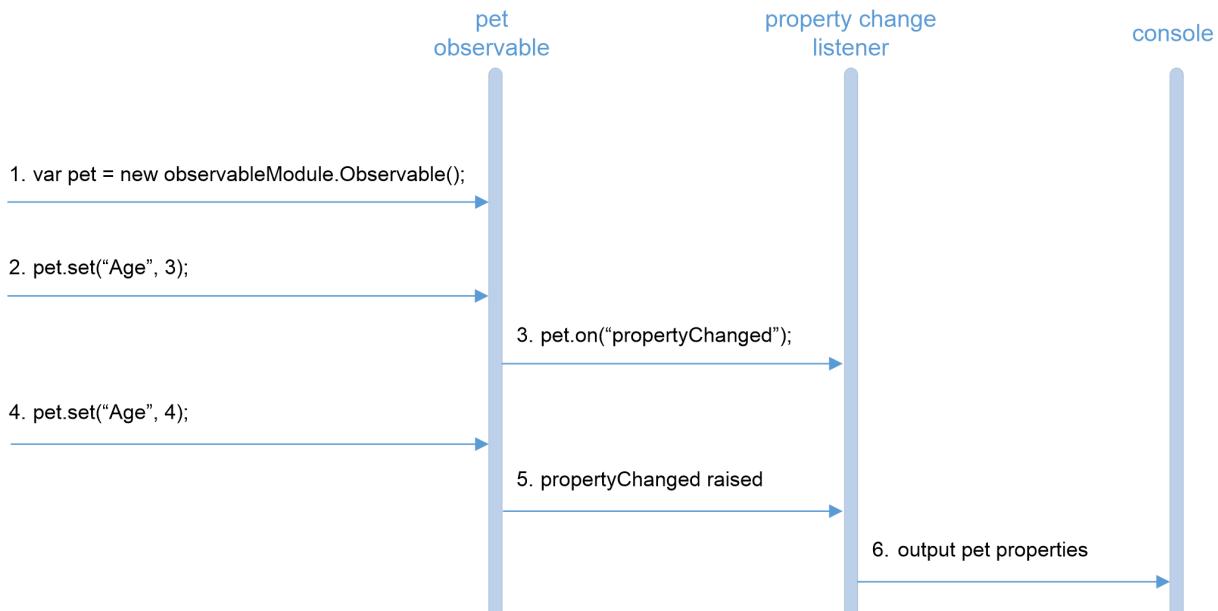


Figure 8.5 A sequence diagram describing an observable's property changed event being raised.

In step 1, the observable module is imported and the pet observable is created.

```
var observableModule = require("data/observable");
var pet = new observableModule.Observable();
```

The observable module exposes an object named *Observable*. At a high level, the Observable object is just like a plain-old JavaScript object (POJO), except it monitors its properties for changes and raises an event when a property is changed.

Step 2 sets the name, type, and age properties of the pet observable using the `pet.set()` syntax. You'll notice that this syntax differs from POJOs because you can't set the value of a property by using `pet.<property name>` directly.

TIP To set a property of an observable, use the `.set()` function.

After we've set several properties, we establish a function that that listens for changes to the observable's properties (step 3). Every observable object exposes a *property change* event that we can listen to and handle when a property on the object changes. When the property changed event is raised, the observable passes useful data to any function handling the event. We've used one of the properties (object), which is a reference to the observable object. Using this reference, the pet's name, type, and age will be displayed in the console.

Step 4 updates the pet's age to 4. When the age is updated, the observable will detect the change and raise the property change event (step 5), which will be handled by our function and print the pet's updated information to the console (step 6).

Go ahead and run the pet scrapbook app and you should see the following results in the console window (figure 8.6).

```
Aug 17 21:16:12 Nicks-MBP-2 observableexample[66936]: CONSOLE LOG file:///app/main-page.js:11:16:
Your pet is a Dog named Riven and is 4 years old.
```

Figure 8.6 The console log message that occurred from the property change event when a property of the pet observable changed.

Observables are just the first step in working with databinding in NativeScript, providing you with an easy way to listen for and respond to changes that happen to an object. You just learned how to respond to changes in an observable object in code, but does this mean you need to write code to respond to every property changed event? Certainly not. When you use observables inside the context of Databinding, NativeScript takes care of handling the property changed events for you automatically. Let's take a look at a basic databinding example.

8.2.1 Property binding

The simplest form of databinding in NativeScript is called *property binding*. Property binding is when you link the property of a UI element with the property of an observable. Once linked, changes made to either the UI element's property or the observable's property will be automatically reflected in both places.

Although you don't need to know about the inner-workings of property binding, we think it's important a you know a little about something special called a *bindable object*.

DEFINITION Bindable objects are UI elements that are inherited from the Bindable class. The Bindable class is special because it allows objects inheriting from it to be data bound with an observable.

Property binding inherently requires you to databind between an observable and a bindable UI element.

NOTE You shouldn't worry about the bindable class, because it's not something you will use directly when developing a NativeScript app. The truth is, you'll use bindable objects all the time (because every UI element is bindable), and you won't really think about the Bindable class. If you want to learn more about the Bindable class, visit the official NativeScript documentation at https://docs.nativescript.org/api-reference/classes/_ui_core_bindable_.bindable.html

Let's take a look at a visual example of how property binding works, then we'll write some code to bind an observable and UI element together. Figure 8.7 shows how several UI elements inherit from the bindable class, and how the bindable class enables the UI elements to data bind with an observable object via the `bind()` method.

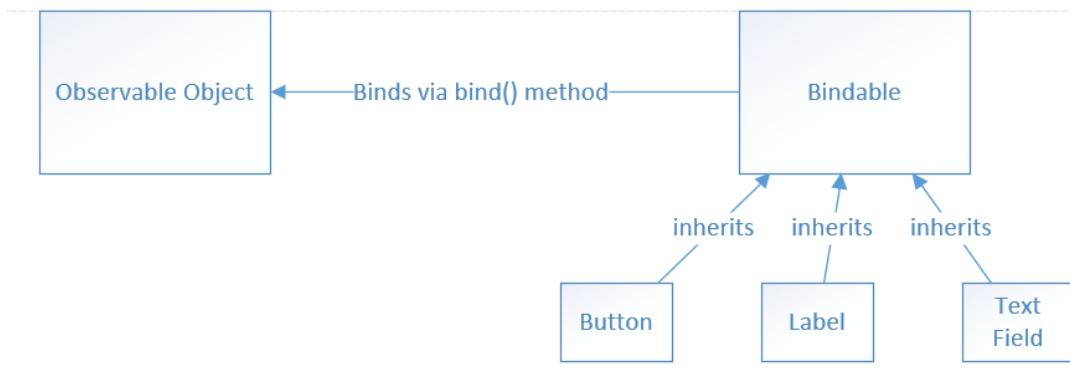


Figure 8.7 Observable object are able to update UI elements derived from the Bindable class because the Bindable class exposes a common interface that observable objects can talk to.

Let's see this in action by binding to the `text` property of a `Label` element. First, update the view of the `home-page.xml` file to include a stack layout and label, as shown in listing 8.2.

Listing 8.2 The views\home-page.xml file

```
<Page loaded="onLoaded"> //#A
<StackLayout>
    <Label id="title" /> //#B
</StackLayout>
</Page>
#A Bind to the loaded event of the page
#B Give the label an id so we can access it in JavaScript
```

Notice that we've given the label an id so we can easily find it in the UI via JavaScript. Next, update the home-page.js file by moving the pet observable declaration into the `onLoaded` function, removing the property change event, and establishing data binding between the label and the pet observable by using the `bind()` function (listing 8.3).

Listing 8.3 The views\home-page.js file showing binding to the text property of a Label element

```
var observableModule = require("data/observable");
var viewModule = require ("ui/core/view");

exports.onLoaded = function(args) {
    var page = args.object;
    var pet = new observableModule.Observable();
    var label = viewModule.getViewById(page, "title"); //#A

    var bindingOptions = {           //#B
        sourceProperty: "Name", //#B
        targetProperty: "text" //#B
    };

    label.bind(bindingOptions, pet); //#C

    pet.set("Name", "Riven");
}
#A Find the label by its id property.
#B The binding option object describes the properties we'll be binding from (source) and to (target).
#C Bind the label and pet observable together using the binding options.
```

When you run the pet scrapbook app, you'll see that the label doesn't look different from a label that is defined with static text: `<Label text="Riven" />`. Even though the label doesn't look different, there's a lot going on in listing 8.2 and 8.3 to make it look the same. To bind the label's `text` property to the `Name` property of the observable, listing 8.3 does two things:

3. A binding object is used to describe the properties we'll be binding together, specifying the `name` property as the source (or from) property and the `text` property as the target (or to) property
4. The label is data bound to the pet observable by calling the `bind()` method.

The `bind()` method accepts binding options and an observable object as parameters and knows how to work with the observable object's `on change` event (this is the same event that you manually implemented earlier) to allow data changes to be reflected in both the pet observable and the label. Figure 8.8 shows the pet scrapbook running.



Figure 8.8 A label that has property binding setup in code.

The first time you see the binding syntax in code, it can be a bit confusing. Let's re-write the bind() function from listing 8.3 to include the binding options directly in the function call: `label.bind({ sourceProperty: "Name", targetProperty: "text" }, pet)`. Figure 8.9 show the relationship between this function call and how it links the properties of the label element and pet observable.

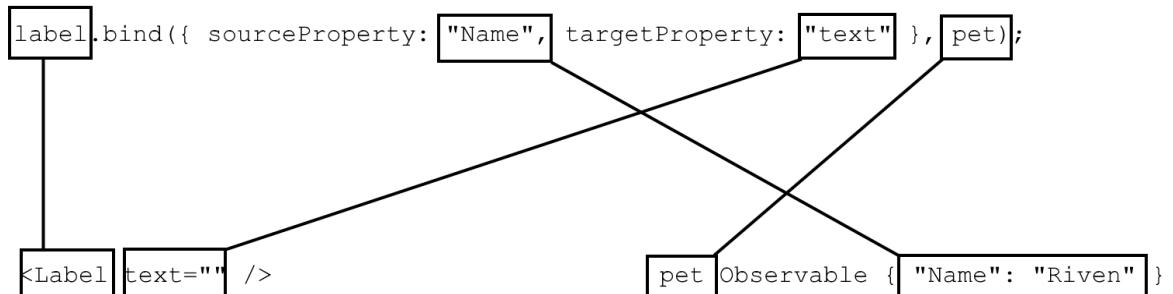


Figure 8.9 The bind() function links together the label UI element and the per observable.

At this point you may be thinking that property binding sure is a lot of work, and why would you spend so much time manually binding observables and UI elements together. And you're right: property binding is a whole lot of work. It turns out you'll rarely need to use property binding, instead you'll use the shortcut version of property binding, called XML binding. Let's take a look at XML binding and how it dramatically reduces the amount of code you write to data bind a UI element and an observable.

One-way databinding versus two-way databinding

Databinding can flow one way or two ways. In one-way data bindings, the data flows from an observable to the UI *or* the data flows from the UI to an observable (but not both).

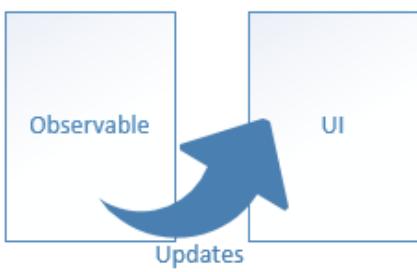
For example, if you use one-way databinding to bind an observable to a text field in the UI, and the user changes the value of the text field, the linked observable will not be updated.

Keeping the same example in mind, using two-way databinding between the text field and an observable, you can update both the text field or the observable and affect a change in the linked property. This is called two-way because it flows data in 2 directions: the UI to the observable and the observable to the UI.

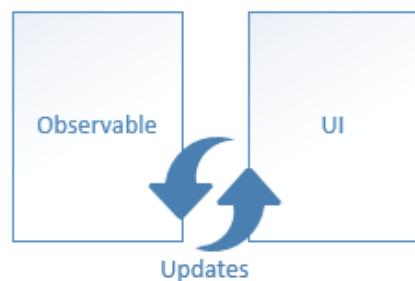
As you will come to learn, two-way databinding is very powerful and helpful to you as a developer.

When working with data in this book I will be showing you two-way databinding.

One-Way Binding



Two-Way Binding



The difference between one-way databinding and two-way databinding.

8.2.2 XML binding

XML binding does the same thing as property binding: it links an observable property and a UI element's property and keeps the two in sync. Even though XML binding acts the same as property binding, it is much easier to configure and requires very little code to set up. Instead of defining the source and target properties in JavaScript, you declare the binding relationship between the UI and an observable in a page's XML markup.

XML binding in NativeScript uses *mustache syntax* to denote the binding.

DEFINITION Mustache syntax is a way to denote a special value using curly braces { and }. The syntax gets its name because the curly brace looks like a sideways mustache. In NativeScript a double curly brace is used.

Let's refactor the home page of the pet scrapbook again to swap out the property binding for XML binding. We'll start by changing the home-page.xml file by removing the id field of the label, and adding a data bound text field (listing 8.4).

Listing 8.4 The views\home-page.xml file updated to use XML binding

```
<Page loaded="onLoaded">
    <StackLayout>
        <Label text="{{ Name }}" /> //#A
    </StackLayout>
</Page>
```

#A XML binding uses double mustache syntax to denote the data bound property

When using XML binding, you don't have to locate a UI element and call the bind() method to link it with an observable. Instead, the double mustache text="{{ Name }}" syntax is used. This syntax is special because it identifies the UI element's and observable's properties that will be data bound. In listing 8.4, the text (UI element) and Name (observable) properties will be data bound together.

Now, simply placing the text="{{ Name }}" syntax in the XML markup isn't enough to make data binding occur. We're missing an important item: what observable will be data bound to? To get the answer, we'll need to refactor our code in the home-page.js file to remove the property databinding and tell the page which observable to use by setting the page's *binding context*.

DEFINITION A UI element's binding context identifies the observable object that will participate in data binding with the UI element. Typically, you set a binding context of a page and create a single observable used for the page. This way, all elements on the page can share the binding context (and observable).

Let's see this in action (listing 8.5).

Listing 8.5 Assigning an observable to the page's binding context on home-page.js

```
var observableModule = require("data/observable");
var viewModule = require ("ui/core/view");

exports.onLoaded = function(args) {
    var page = args.object;
    var pet = new observableModule.Observable(); #A

    page.bindingContext = pet; #B

    pet.set("Name", "Riven");
}

#A The pet object is an observable that will bind to all elements on the page
#B Setting pet to the page's binding context establishes it as the page-level observable used for binding
```

As you'll see in listing 8.5, we've removed the property binding code and replaced it with code to specify a binding context. The binding context identifies the observable that the page uses in data binding.

Now that you've learned about XML data binding, let's turn our attention to using XML data binding to build the pet scrapbook app.

BUILDING THE HOME PAGE

The pet scrapbook starts with a home page where a user can navigate to an about page or continue on to the contents of the scrapbook. We'll be reusing the *home-page.xml* and *home-page.js* files you created in the previous sections.

Starting with the UI, we'll add data bound labels for a title and footer, an image, and two buttons. As you can see in listing 8.6, the data-bound labels use the mustache syntax. We chose to use XML binding for these fields because the header and footer text isn't something we want hard-coded in our app.

Listing 8.6 The views\home-page.xml showing XML binding

```
<Page loaded="onLoaded">
<ScrollView>
    <StackLayout>
        <Label class="header" text="{{ header }}" /> // #A
        <Image src "~/images/home.png" />
        <Label class="footer" text="{{ footer }}" /> // #A
        <StackLayout orientation="horizontal" horizontalAlignment="center" >
            <Button class="marginRight" text="About" />
            <Button class="marginLeft" text="Continue" />
        </StackLayout>
    </StackLayout>
</ScrollView>
</Page>
```

#A Binding the text property of a label to a property defined on an observable object

As you'll recall from earlier in this chapter, using mustache syntax to data bind the header and footer is only one-half of the equation. If we were to run the Pet Scrapbook app, the header and footer would be blank, because the page doesn't yet know which observable to use in data binding. Listing 8.7 sets up an observable object and sets it to the page's binding context in the *home-page.js* file.

Listing 8.7 The views\home-page.js file showing the implementation of an observable for the home page

```
var observable = require("data/observable");

exports.onLoaded = function(args) {
    var page = args.object;
    var home = new observable.fromObject({
        header: "Pet Scrapbook",
        footer: "Brosteins ©2016"
    });
    page.bindingContext = home; // #A
};

#A Set the binding context of the home page to the observable we created so it can access the properties in our UI via mustache syntax
```

Add the following styles to the *home-page.css* file (listing 8.8), and run the pet scrapbook.

Listing 8.8 The views\home-page.css file adding style to the home page

```
.header {
```

```
    font-size: 32px;  
}  
label {  
    text-align: center;  
    margin-top: 10px;  
    margin-bottom: 10px;  
}  
.footer {  
    font-size: 10px;  
}
```

After running the pet scrapbook (figure 8.10), you won't notice that the labels are data bound; however, we have made our app more dynamic because the values for the header and footer are no longer hard-coded in the XML view code.

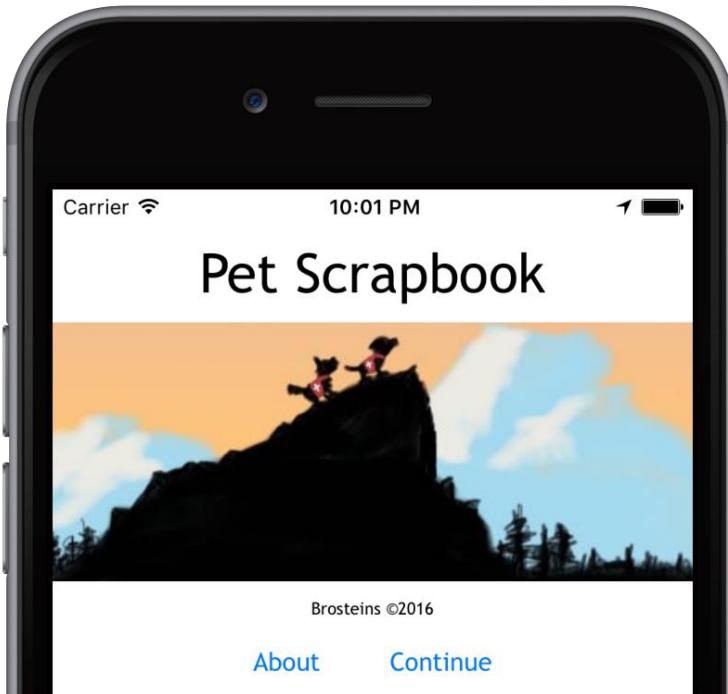


Figure 8.10 The Pet Scrapbook home page updated to use databinding for the title and footer.

Although data binding the home page fields may feel a bit artificial at this point, we'll continue to use these same techniques throughout the book, and you'll see how powerful it can be.

TYING IT ALL TOGETHER

Before we move on, let's break down what's happening on this page and visualize how each label was data bound. In listing 8.7, we established an observable object with two fields: header and footer. By using the `fromObject()` method, we can create an observable on-the-fly.

```
var home = new observable.fromObject(
  { header: "Pet Scrapbook", footer: "Brosteins ©2016" });
```

We then set the page's binding to the home observable object: `page.bindingContext = home`. Last, we used mustache syntax, to define the properties of the observable object in the XML code of the Home page.

```
<Label text="{{ header }}" />
<Label text="{{ footer }}" />
```

With these three components working together, the home page places *Pet Scrapbook* into the header and *Brosteins ©2016* into the footer (figure 8.11).

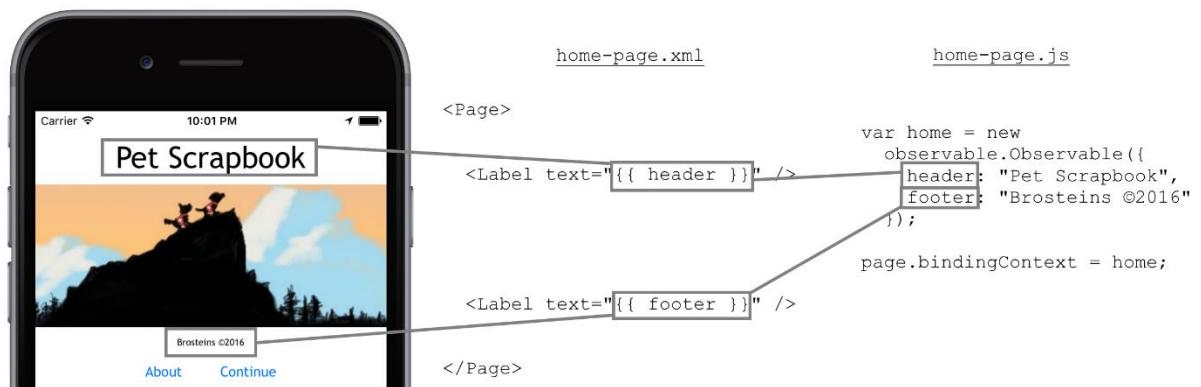


Figure 8.11 The home page data binding the header and footer labels with mustache syntax and an observable

8.2.3 Pet scrapbook page

Databinding is not just limited to labels; in fact, you will probably want to bind to additional controls inside your app so your app is more dynamic. Let's take a look and see how you can implement databinding on different UI elements by implementing a scrapbook page in the Pet Scrapbook app.

The scrapbook page will allow users to create a memory of one of his or her pets. In the first iteration of the scrapbook page we will allow users to enter the following information (as we progress further in this chapter we will update the scrapbook page to allow multiple pets):

- Title
- Gender
- Birthday

Let's get started implementing the scrapbook page by adding a *scrapbook-page.xml* and *scrapbook-page.js* file to the *views* folder of the Pet Scrapbook app. We will be adding gender and birthday fields to the Scrapbook page first so users can enter some identifying information about their pet. The gender and birthday fields will introduce you to two new UI elements: the *ListPicker* element and the *DatePicker* element. Listings 8.9 and 8.10 show an example of how to define a list picker and date picker in XML and bind the list picker to an array of items.

DEFINITION ListPicker is a user interface element used to display a selectable list of values to the user. To create a ListPicker element, use the XML code <ListPicker>...</ListPicker>.

DEFINITION DatePicker is a user interface element used to display a selectable month, day, and year to the user. To create a DatePicker element, use the XML code <DatePicker>...</DatePicker>.

Listing 8.9 The views\scrapbook-page.xml with a list picker and date picker element

```
<Page loaded="onLoaded" >
  <StackLayout>
    <DatePicker />
    <ListPicker items="{{ items }}" /> //#A
  </StackLayout>
</Page>
#A The list picker's items are data bound to the items object of the page's binding context
```

Listing 8.10 Binding items to the ListPicker element on the views\scrapbook-page.js

```
var observable = require("data/observable");

exports.onLoaded = function(args) {
  var page = args.object;
  var listItems = new observable.Observable();          //#A
  listItems.items = ["Item 1", "Item 2", "Item 3"]; //#A

  page.bindingContext = listItems;                      //#B
}
#A An array of three items is set to the items property of the listItems observable
#B The page's binding context is set to the observable
```

Figure 8.12 shows the DatePicker and ListPicker elements rendered on native controls on an iOS device.

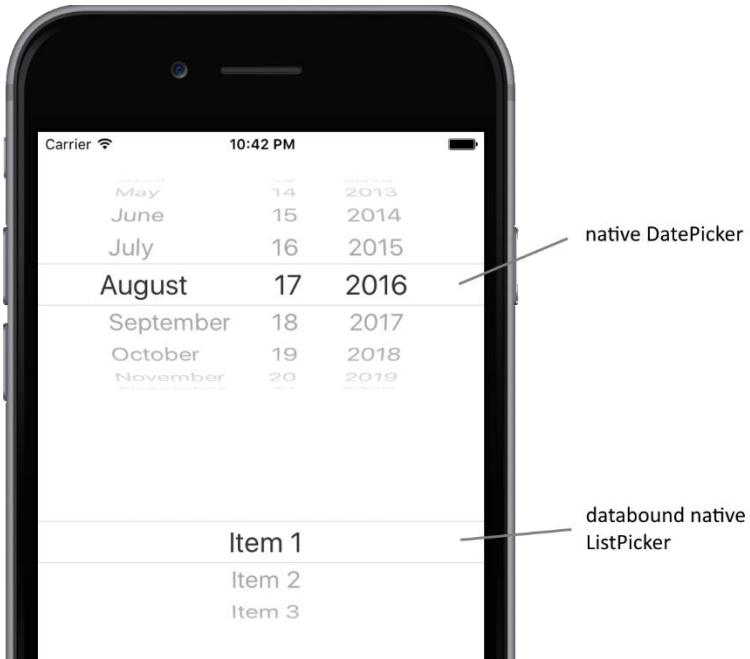


Figure 8.12 A native DatePicker element and ListPicker element rendered on an iOS device. The ListPicker element is bound to a selectable list of items.

As you can see the DatePicker and ListPicker elements work the same way that other elements do that we have discussed up to this point. The one difference you may have noticed is with the ListPicker element. The ListPicker element's *items* property doesn't bind to just a single value, but instead expects to bind to an array of items.

Now that you have been introduced to a couple of new controls, let's continue finishing the Scrapbook page by adding a title and descriptive labels. We will be using XML binding with all of these controls so that we can access the data in JavaScript (because eventually we will want to save the scrapbook page the user has created). Listing 8.11 puts together the new controls (and an old one) alongside databinding to implement the scrapbook page.

Listing 8.11 The views\scrapbook-page.xml showing more databinding techniques

```
<Page loaded="onLoaded">
    <StackLayout>
        <Label text="Title: " />
        <TextField class="header" text="{{title}}"/>
        <Label text="Age: " />
        <DatePicker date="{{ date }}"/>
        <Label text="Gender: " />
        <ListPicker items="{{ genders }}" selectedIndex="{{ gender }}"/> //##A
        <Button tap="onTap" text="Done" />
```

```

        </StackLayout>
    </Page>
#A Added data binding to the selected index of the list picker so we know which list item is selected

```

As you can see in listing 8.11, we have added several XML bindings for the DatePicker, TextField, and ListPicker elements. With the additional bindings that we have added we will be able to access the values that the user of the application selects or types into these fields. Pay special attention to the date picker's `date="{{ date }}"` property and list picker's `selectedIndex="{{ gender }}"` property. The DatePicker element provides a property named `date` that can be data-bound to capture the selected date. Similarly, you can bind to the `selectedIndex` property of a ListPicker to get the selected index.

Now that we have our UI organized, add the data binding configuration code as shown in listing 8.12.

Listing 8.12 the views\scrapbook-page.js showing how to setup data binding

```

var observable = require("data/observable");

exports.onLoaded = function(args) {
    var page = args.object;
    var scrapbook = new observable.fromObject({ //#A
        genders: ["Female", "Male", "Other"] //#A
    }); //#A

    page.bindingContext = scrapbook;
};
#A fromObject() is a shortcut that creates an observable from a JavaScript object

```

We've seen this before, and the code doesn't look new, with two exceptions. First, we used the `fromObject()` method to create an observable from a JavaScript object. It's a shortcut, you should use it.

TIP The `fromObject()` method is a shortcut for creating an observable from an existing JavaScript object.

Second, you'll notice that the scrapbook observable didn't specifically define the `gender` and `date` properties that were bound to the picker elements. Because of JavaScript's dynamic nature, you don't have to specify them (but if you do, it won't hurt).

ACCESSING DATA-BOUND INFORMATION

Great work! But, now that we have the UI data bound, how can you access the properties? There's several ways to do this, and you'll learn one of them right now; we'll point out a second way later in the chapter. Add the done button's tap event handler to the `scrapbook-page.js` file (listing 8.13)

Listing 8.13 Accessing data bound properties on the views\scrapbook-page.js

```

exports.onTap = function(args) {
    var page = args.object; // #A
    var scrapbook = page.bindingContext; // #B

    console.log("You have made " + scrapbook.title); // #C
    console.log("Age: " + scrapbook.date.toLocaleDateString()); // #C

```

```
        console.log("Gender selected:" + scrapbook.genders[scrapbook.gender]); //#C
    }
#A args.object is a reference to the page object
#B bindingContext variable contains a reference to the scrapbook observable
#C After the user has entered values for into the UI controls, you can access the properties in JavaScript because the fields are databound
```

In listing 8.13, we are using a button tap event to access the binding context the scrapbook page. When a tap event is handled, the handler passes a reference to the page via the object property, so args.object is a reference to the page. With a reference to the page, the bindingContext property has a reference to the original observable, which contains all the data bound properties.

WARNING Watch out for the list picker's selectedIndex property: it is the selected index, not the selected list item. To get access to the actual selected list item value, you'll need to do a little work and lookup the list item value at the selected index.

When run, the Scrapbook page logs the values of these properties to the console, as seen in figure 8.13.

```
CONSOLE LOG file:///app/views/scrapbook.js:17:16: You have made Riven's Page
CONSOLE LOG file:///app/views/scrapbook.js:18:16: Age: September 15, 2022
CONSOLE LOG file:///app/views/scrapbook.js:19:16: Gender selected:2
```

Figure 8.13 The output of the console logging of the values that were data-bound on the scrapbook page.

Although logging values to the console isn't all that interesting it is important to understand that you have access to the all the properties of an observable that is set as the binding context of a page. In later chapters and examples, we will implement more complex business logic and manipulate the properties that are data bound. For now, let's take a look at something more interesting that you can do with databinding: binding expressions.

8.2.4 Binding Expressions

Wouldn't it be nice if the pet's name were displayed in the page's title, but with some text added on to the end, like "Riven's Scrapbook Page?" Now, think about how you'd do that with the tools you have at your disposal. So far, you've only learned how to data bind to the entirety of a field; that is, replace the entire contents of a label's text property with a data-bound value.

So, how could you display "Riven's Scrapbook Page"? One option is to use two labels: one data-bound to the title, and a second with static text of *Scrapbook Page*. Yuck!

A second option is to use a single label data-bound to the title, while incorporating a binding expression.

DEFINITION A binding expression is logic or a calculation that applied to an XML binding directly in the UI, allowing you to transform the data-bound value displayed.

Many basic logic operators are available for use within expression bindings. For the Pet Scrapbook app, we'll use a unary operator to concatenate two strings together to create "Riven's Scrapbook Page" with a single label.

NOTE For a full list of all supported operators that you can use in binding expressions, please view the official NativeScript documentation at <https://docs.nativescript.org/core-concepts/data-binding#supported-expressions>.

Binding expressions extend the mustache syntax. When using a binding expression, we use a comma to delimit the property that we want to bind to and the expression (with the property coming first). Listing 8.14 shows how to create a binding expression that concatenates two strings together. Change the first Label element in the *scrapbook-page.xml* file to use a binding expression:

Listing 8.14 The *views\scrapbook-page.xml* updated with a binding expression that concatenates two strings together

```
<Page loaded="onLoaded">
  <StackLayout>
    <Label text="{{ title, title + ' Scrapbook Page' }}" /> // #A
    <TextField class="header" text="{{ title }}" hint="Enter title..."/>
    <Label text="{{ 'Age: ' + calcAge(year, month, day) + ' years old'}}" />
    <DatePicker year="{{ year }}" month="{{ month }}" day="{{ day }}" />
    <Label text="Gender: " />
    <ListPicker items="{{ genders }}" selectedIndex="{{ gender }}" />
    <Button tap="onTap" text="Done"/>
  </StackLayout>
</Page>
```

#A Adding a binding expression to concatenate a static string and a data-bound string together in the label field

Remember, the first field of a binding expression (the value before the comma) is the data-bound property. Once the property is declared before the comma, it can be used in the binding expression, which follows the comma. In listing 8.4, the binding expression displays the title field concatenated with *Scrapbook Page*.

Run these updates in your emulator and enter your pet's name in the text field. You'll notice the header will dynamically change as you type. You should see something like figure 8.14.

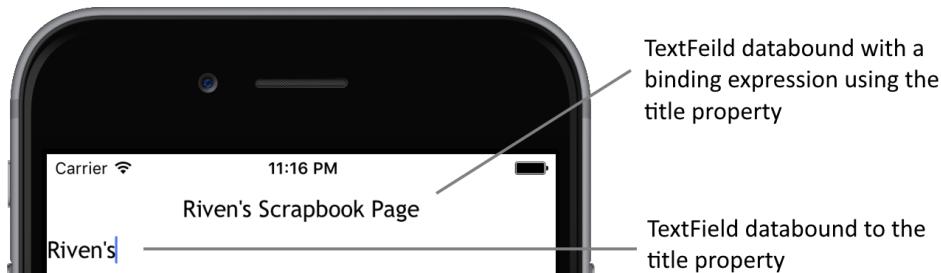


Figure 8.14 A binding expression showing string concatenation. The expression is applied to a label that binds to the

same property as the text field.

One important note is that you can use the same XML binding for multiple controls. As you can see from figure 8.14, you are able to bind the title of the scrapbook page that a user has enter into a text field to a label at the same time. When you change the value of the text field you will notice that the label changes at the same time as well! NativeScript is handling all the property changed events for you to set the values in both of the UI elements.

BINDING EXPRESSION FUNCTIONS

Binding expressions can be much more advanced than just simple string concatenation. In fact, you can have a binding expression that uses any function that you define in JavaScript. Let's look back at the age field of the scrapbook page. The age is data-bound to a date picker's date property, which looks a little strange.

```
<Label text="Age: " />
<DatePicker date="{{ date }}" />
```

If you pay close attention, you'll notice that the pet's age is described as a date, which isn't right. An age is described as a single number, not a date. Ideally, I'd like to change the UI by adding a data-bound label to display the pet's age based upon the birth date selected. Dynamically calculating the pet's age sounds like a great use case for a binding expression function.

Listings 8.15 and 8.16 show the update XML code and JavaScript code needed to implement a custom age calculation function and display it on the UI using a binding expression.

Listing 8.15 The views\scrapbook-page.xml file updated with a custom function in a binding expression

```
<Page loaded="onLoaded">
  <StackLayout>
    <Label text="{{ title, title + ' Scrapbook Page' }}" />
    <TextField class="header" text="{{ title }}" hint="Enter title..."/>
    <Label text="{{ 'Age: ' + calcAge(date) + ' years old'}}" /> //##A
    <DatePicker date="{{ date }}" />
    <Label text="Gender: " />
    <ListPicker items="{{ genders }}" selectedIndex="{{ gender }}" />
    <Button tap="onTap" text="Done"/>
  </StackLayout>
</Page>
#A Calling a custom function on the observable object that performs an age calculation. You can pass data-bound properties or other values to a custom function. You can also combine the custom function with a binding expression.
```

Listing 8.16 The views\scrapbook-page.js file showing a function calculating age and used in a binding expression

```
exports.onLoaded = function(args) {
  var page = args.object;
  var scrapbook = new observable.Observable({
    genders: ["Female", "Male", "Other"],
    calcAge: function(birthDate){ //##A
      var now = Date.now();
```

```
        var diff = Math.abs(now - birthDate) / 1000 / 31536000; //#B
        return diff.toFixed(1);
    }
});

page.bindingContext = scrapbook;
};

#A Implementing a function on the observable object that is available to use in the UI with XML databinding
#B Math.abs() returns the number of milliseconds different, so we divided by 1000 to get seconds, then by the
number of seconds in a year (265 * 24 * 3600 = 31526000) to get years
```

When setting up the binding expression for the title of the Scrapbook page we just used a unary operator. When using a custom binding expression, we can bind to a property on our observable that is a custom function instead of just a simple type. NativeScript even handles passing the parameters into the method for you.

TIP You can combine binding expressions with custom functions for more flexibility within your UI. Just make sure the function is a property on your observable.

Figure 8.15 shows the resulting scrapbook page after updating it with a more complex binding expression.

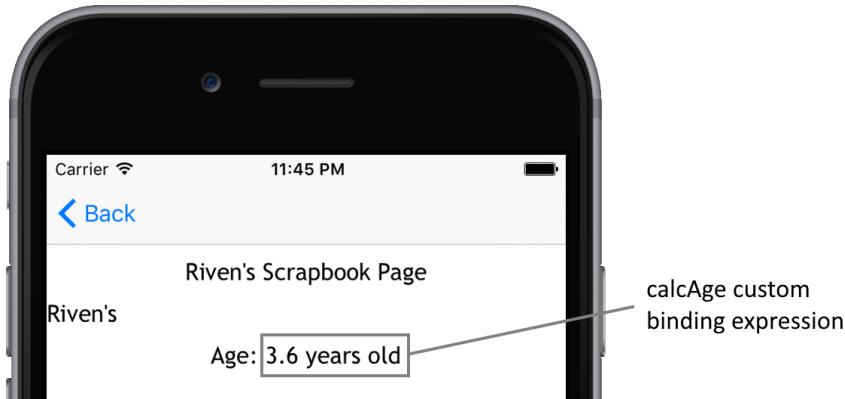


Figure 8.15 The age label field is bound to a custom binding expression that calculates the age of the pet based on the birthday entered.

Hopefully, you have started to see how databinding is going to assist you while you develop NativeScript apps. Now that you have a solid foundation on the basics of observables and databinding, it is time to tackle binding to arrays.

8.3 Observable arrays

Binding to arrays isn't much different than binding to a single object. In fact, we've already bound to the genders array to display a list of available gender options: `genders: ["Female", "Male", "Other"]`. Just like this example, you typically bind to an array when you want to display (or select from) a list of values. But what if one of the values in the data-bound array changes? Will the data-bound UI element automatically update? It depends. Take the following code snippet as an example:

```
var scrapbook = new observableModule.fromObject({
  genders: ["Female", "Male", "Other"]});
```

If you define your array as a property of an observable, the observable will only notify its data-bound partner that the observable has changed if the entire array is replaced. Replacing an array of 3 items may not seem like a lot of work, but imagine the array had 100 items, or 1,000 items: that is an awful lot of inefficient work to do (replacing the entire array object), just to get the observable to report a data change. That's why there's a special observable object called an observable array.

DEFINITION Observable arrays are data objects (like observables) that provide your code with notifications when an item is added, removed, or changed from an array.

Creating and using an observable array is very similar to the observable object you learned earlier, except it requires a different NativeScript core module. Listing 8.17 shows how to access the observable module and create an observable array.

Listing 8.17 Creating an observable array

```
var observable = require("data/observable-array");

var pets = new observable.ObservableArray("Riven", "Pittens"); //#A
pets.push("Nibbles"); //#B
#A You can initialize an observable array by passing in items to the ObservableArray() function
#B Additional items can be added to the array
```

As you can see from listing 8.17, observable arrays work just like a normal array in JavaScript (you can push and splice them as needed). When you add and remove items to the array then the `propertyChange` event of the observable will fire. The property change event for an observable array works just like a regular observable's event, so we're not going to go into detail. Instead, let's take a look and see how we can make the observable array work inside of the Pet Scrapbook app.

8.3.1 Using an observable array to build master-detail pages

So far, the Pet Scrapbook app isn't much of a scrapbook because it only has a single scrapbook page and users can only enter information for a single pet. Let's refactor the scrapbook so users can add information for multiple pets.

Over the next several sections, we'll be changing the main page of the app to show a list of all the scrapbook pages. From the main page, users will be able to add new pages or tap one of the existing pages to view the details. These changes will form the foundation for a standard UI design you'll use over and over: master-detail.

DEFINITION Master-detail UI pattern describes a relationship (and navigation) between two pages and a collection of data. The master page contains a brief summary of many data points, and a user can navigate to a separate page displaying the detailed view of each data point.

In the Pet Scrapbook app, the data points we'll be tracking are scrapbook pages. The master page will be a listing of each page, with the detail pages displaying all of the information (pet's name, age, gender, etc.).

To build the master-detail pages, we'll be using an observable array to hold the scrapbook pages. The observable array will be data bound to a new UI element, the *list view*, to display the list of scrapbook pages on the master page.

DEFINITION The list view is a UI element used to display a list of items. The ListView element supports a templating system so you can create a complex collection of UI elements that is displayed for each item in the list. To create a ListView element, use the XML code `<ListView>...</ListView>`.

Although list views can be data bound to simple arrays, data binding to an observable array is ideal because the list view's UI is automatically updated as items are added, removed, and updated in the observable array. Listing 8.18 shows the an updated *scrapbook-page.xml* page with a list view. The list view on this page will eventually allow us to add multiple pets to our scrapbook.

Listing 8.18 The `views\scrapbook.xml` page updated with a list view

```
<Page loaded="onLoaded">
    <StackLayout>
        <ListView items="{{ pages }}" itemTap="onItemTap"> //##A
            <ListView.itemTemplate> //##B
                <StackLayout>
                    <Label text="{{ title, title + ' Scrapbook Page' }}"/> //##C
                </StackLayout>
            </ListView.itemTemplate>
        </ListView>
    </StackLayout>
</Page>
#A The pages property is an observable array of scrapbook pages
#B All content defined inside of an item template will be rendered once for each item in the pages observable array
#C The title property binds to a property on each array item
```

NOTE The list view template can only have one child property. If you intend to have multiple UI elements displayed for each item, you'll need to place them inside a layout container.

Notice that we have added a list view to this page and a template for the list view. The initial databinding for the list view is set via the `items` property. In listing 8.18, an observable array named `pages` contains all of our scrapbook pages. For each item in the data bound array, the contents of the item template will be displayed in the UI. When each item is displayed, any data-bound UI elements in the item template (like the label data-bound to the `title` property) will be bound to each item of the array.

Now that we have updated the UI code, we still need to update our JavaScript code to add an observable array names *pages* (listing 8.19).

Listing 8.19 The views\scrapbook-page.js updated to bind data to the list view

```
var observable = require("data/observable");
var observableArray = require("data/observable-array");

exports.onLoaded = function(args) {
    var page = args.object;
    var calcAge = function(year, month, day){
        var date = new Date(year, month, day);
        var now = Date.now();
        var diff = Math.abs(now - date) / 1000 / 31536000;

        return diff.toFixed(1);
    }
    var genders = ["Female", "Male", "Other"];
    var emptyScrapbookPage = new observable.Observable({
        genders: genders,
        calcAge: calcAge
    });
    var filledScrapbookPage = new observable.Observable({
        genders: genders,
        title: "Riven's Page",
        calcAge: calcAge,
        gender: 0
    });
    var scrapbook = new observable.Observable({
        pages: new observableArray.ObservableArray(emptyScrapbookPage,
filledScrapbookPage)
    }); //#A

    page.bindingContext = scrapbook;
};

#A The binding context for the scrapbook now contains an observable array. Observables can contain other observables or objects that are not observable.
```

When you run this example, you'll see the updated Scrapbook page that has added an observable array containing two pages (figure 8.16).

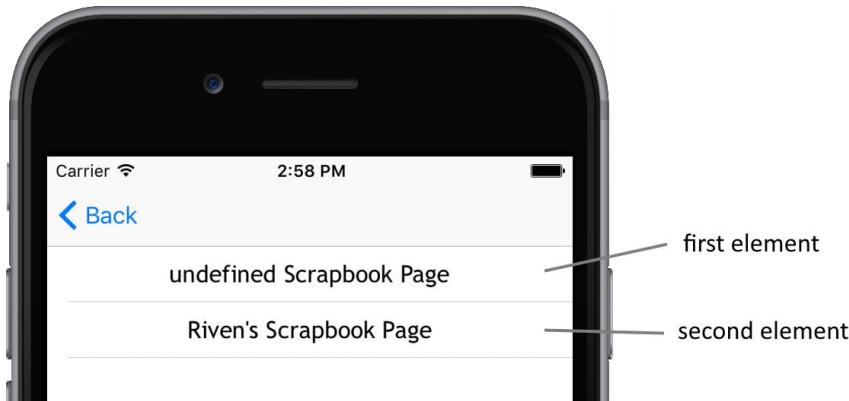


Figure 8.16 The resulting ListView element that has been databound to an observable collection with 2 elements in it. A list view template is used to render each item in the collection.

NOTE Depending on the emulator device version that you are running on, you may have to scroll your view to see both observables that have been bound to the observable array.

You may have noticed, there is no way for the user to update the pages that are represented by the items in the master page list view. We'll need to update the Pet Scrapbook so that when a user taps on an item in the list view, we navigate to a detail page where the data can be updated. To respond to tapping a list item, we'll use the `itemTap` event of the `ListView` element. In the event handler, we'll navigate to the detail page, passing along our observable array and the index of the array we'd like to view. Update the `scrapbook-page.js` to code in listing 8.20. Don't worry if the changes seem overwhelming: we'll walk you through each of them.

Listing 8.20 The `views/scrapbook-page.js` updated to implement the tap event of the list view

```
var observable = require("data/observable");
var observableArray = require("data/observable-array");
var frame = require("ui/frame");

function scrapbookPageModel(){ // #A
    var model = new observable.Observable();

    model.genders = ["Female", "Male", "Other"];
    model.calcAge = function(year, month, day){
        var date = new Date(year, month, day);
        var now = Date.now();
        var diff = Math.abs(now - date) / 1000 / 31536000;

        return diff.toFixed(1);
    };

    return model;
}
```

```

exports.onLoaded = function(args) {
    var page = args.object;
    var scrapbook;

    if(page.navigationContext != null) { // #B
        scrapbook = page.navigationContext.model;
    }
    else {
        scrapbook = new observable.Observable({
            pages: new observableArray.ObservableArray(new scrapbookPageModel())
        });
    }

    page.bindingContext = scrapbook;
};

exports.onItemTap = function(args) { // #C
    var page = args.object;
    var scrapbook = page.bindingContext;

    frame.topmost().navigate({
        moduleName: "views/scrapbookUpdate-page",
        context: { model: scrapbook, index: args.index } //## D
    });
}

#A Define a reusable model for the each scrapbook page
#B Check to see if this is the first time the page was loaded or not
#C Implementation of the list item tap event handler
#D Pass the scrapbook model and index of the page we want to update to the page we are navigating to

```

We made several changes to the JavaScript code for the scrapbook page. The first thing we added was the `scrapbookPageModel()` function. The next thing that we did is define a function is used as a model for each scrapbook page in the array. This is important because it helps to make our code more maintainable as we continue adding features to the Pet Scrapbook app. Next, we have updated the load event of the page to check if we have data in the navigation context when the page is loaded. This object will be null the first time we load the page (more on this in just a moment when we define the update page). The last item that we have updated on the scrapbook page is the implementation of the tap event for a list view item. When an item is tapped, we navigate to the update page and use the navigation context to pass the scrapbook model and the index of the item that was clicked to the new view. Listings 8.21 and 8.22 show the definition of the update page.

Listing 8.21 The views\scrapbookUpdate-page.xml page

```

<Page loaded="onLoaded"> // #A
<StackLayout>
    <Label text="{{ title, title + ' Scrapbook Page' }}" />
    <TextField class="header" text="{{ title }}" hint="Enter title..."/>
    <Label text="{{ 'Age: ' + calcAge(year, month, day) + ' years old'}}" />
    <DatePicker year="{{ year }}" month="{{ month }}" day="{{ day }}" />
    <Label text="Gender: " />
    <ListPicker items="{{ genders }}" selectedIndex="{{ gender }}" />
    <Button tap="onDoneTap" text="Done"/>
</StackLayout>

```

```
</Page>
#A Move the page model definition to its own xml view
```

Listing 8.22 The views\scrapbookUpdate-page.js file implementing the update code in a separate view

```
var observable = require("data/observable");
var frame = require("ui/frame");
var scrapbook; // #A

exports.onLoaded = function(args) {
    var page = args.object;
    var index = page.navigationContext.index;
    scrapbook = page.navigationContext.model

    page.bindingContext = scrapbook.pages.getItem(index); // #B
};

exports.onDoneTap = function(args) {
    var page = args.object;

    frame.topmost().navigate({ // #C
        moduleName: "views/scrapbook-page",
        context: { model: scrapbook } // #C
    });
};

#A Store a reference to the scrapbook model so it can be sent back later
#B Set the binding context of the page to the page that was selected from the list view
#C Set the navigation context to the scrapbook when done is tapped
```

When an item is tapped in the list view, the page in listings 8.21 and 8.22 is loaded and shown to the user. When the update page is loaded, we use the data that is passed in the navigation context and bind the appropriate model to the update page by using the *index* of the tapped item.

Now that we have properly handled the *itemTap* event, we are able to update and change the information of each scrapbook page; however, we still have an issue because the user can't add more scrapbook pages to the scrapbook! Let's solve this problem and learn about another UI element called the *action bar*.

DEFINITION The action bar is a UI element used to display a header in an application. The *ActionBar* element is generally used to display a title as well and other controls in an application. To create an *ActionBar* element, use the XML code `<ActionBar>...</ActionBar>`.

We will be using the action bar to allow users to add new pages to the scrapbook. Let's take a look and see how to implement the action bar.

8.4 Action bar

The *ActionBar* element is going to help us finish polishing off the Pet Scrapbook app. You are probably already familiar with the action bar concepts from apps that you may use on your own device (figure 8.17 shows the common action bar pattern that many mobile apps use); the action bar serves the dual purpose

or displaying information at the top of the app and containing controls or buttons for the user. In the case of the Pet Scrapbook app, we'll use the action bar to display the title of the app and a button to add additional scrapbook pages.

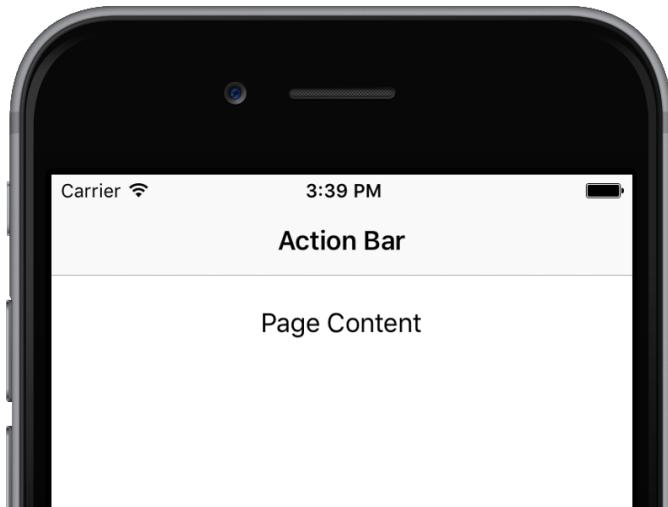


Figure 8.17 The common layout of a mobile app showcasing the location of an action bar versus the content of a page.

We will implement the action bar on the main page of the scrapbook app where we have the list of scrapbook pages. Listing 8.23 shows the updated `scrapbook-page.xml` file to with the newly included `ActionBar` element.

Listing 8.23 The `views\scrapbook-page.xml` file updated to include an action bar

```
<Page loaded="onLoaded">
    <Page.actionBar> // #A
        <ActionBar title="Pet Scrapbook" > // #B
            <ActionItem tap="onAddTap" ios.position="right" // #C
                text="Add" android.position="actionBar"/> // #C
        </ActionBar>
    </Page.actionBar>
    <StackLayout>
        <ListView items="{{ pages }}" itemTap="onItemTap">
            <ListView.itemTemplate>
                <StackLayout>
                    <Label text="{{ title, title + ' Scrapbook Page' }}"/>
                </StackLayout>
            </ListView.itemTemplate>
        </ListView>
    </StackLayout>
</Page>
#A Defining an ActionBar element
#B Setting the title of the ActionBar element
#C Adding a control to the ActionBar element
```

#C Setting the position and text of the ActionBar element on iOS and Android

In listing 8.23 we have added an action bar to the page. Notice that an action bar can have multiple children. The children of the ActionBar element can be labels, buttons, or in this case *action items*.

DEFINITION An action item is a UI element that is used as a control inside an ActionBar element. Action items act much like a button but must be placed inside of an action bar. To create an ActionItem element, use the XML code <ActionItem />.

Although you may be tempted to use a button, there are advantages to using an action item inside of an action bar. The advantage to using an action item over a button is that specific properties pertaining to an action are exposed by the action item. For example, in listing 8.23 we created an action item inside of our action bar and set specific position properties for iOS and Android.

NOTE The action bar is a good example on how cross-platform mobile app development can vary significantly by platform. On the Android platform, there is a concept of an action bar and an options (overflow) menu. The Android options menu is an automatic area that Android places action items that can't fit into the action bar. On iOS, the concept of an options (overflow) menu don't exists. Because of these differences, there is no one-way to define the placement of action items.

Tables 8.1 and 8.2 show the other positions (and the default positions) that we could have used for to position the action item within the action bar.

Table 8.1 Action item positioning options for Android

Position	Description
actionBar (default)	The action item is placed in the Android action bar
popup	The action item is placed in the options (overflow) menu
actionBarIfRoom	The action item is placed in the action bar if there is room else it will be placed in the options menu

Table 8.1 Action item positioning options for iOS

Position	Description
left (default)	The action item is placed on the left side of the action bar
right	The action item is placed on the right side of the action bar

When defining the action item, we used databinding to bind the tap event the same way we would normally do with a regular button. Listing 8.24 shows the updated code to handle the on tap event of the action item so users can add new scrapbook pages to their scrapbook.

Listing 8.24 The views\scrapbook-page.js file updated to handle the tap event of an action item

```
var observable = require("data/observable");
var observableArray = require("data/observable-array");
var frame = require("ui/frame");

function scrapbookPageModel() {
    var model = new observable.Observable();

    model.genders = ["Female", "Male", "Other"];
    model.calcAge = function(year, month, day) {
        var date = new Date(year, month, day);
        var now = Date.now();
        var diff = Math.abs(now - date) / 1000 / 31536000;

        return diff.toFixed(1);
    };

    return model;
}

exports.onLoaded = function(args) {
    var page = args.object;
    var scrapbook;

    if(page.navigationContext != null) {
        scrapbook = page.navigationContext.model;
    }
    else {
        scrapbook = new observable.Observable({
            pages: new observableArray.ObservableArray(new scrapbookPageModel())
        });
    }

    page.bindingContext = scrapbook;
};

exports.onAddTap = function(args) { // A
    var page = args.object;
    var scrapbook = page.bindingContext;

    scrapbook.pages.push(new scrapbookPageModel()); // B

    frame.topmost().navigate({
        moduleName: "views/scrapbookUpdate-page",
        context: { model: scrapbook, index: scrapbook.pages.length - 1 } // C
    });
};

exports.onItemTap = function(args) {
    var page = args.object;
    var scrapbook = page.bindingContext;

    frame.topmost().navigate({
        moduleName: "views/scrapbookUpdate-page",
        context: { model: scrapbook, index: args.index }
    });
};

#A Implement the event handler for action item on tap
```

#B Add a new scrapbook page to the scrapbook
#C Pass data to the update view

Figure 8.18 shows what the ActionBar elements looks like inside the Pet Scrapbook app.



Figure 8.18 The Pet Scrapbook app showing an action bar with an action item button.

Just like a normal button, you can bind to the tap event with an action item. When the action item is tapped we go ahead and create a new scrapbook page and add it to our observable array. In order to pass data to the update page, we use the navigation context. When passing data to the update view we pass the scrapbook and the index of the page that user wants to update within the array.

8.5 Summary

In this chapter, you learned how to do the following:

- Implement databinding using multiple techniques
- Use the action bar with an action item to create a header in an app
- Use a ListView element and a list view item template to simplify your view code
- Use databinding to create a dynamic UI

8.6 Exercise

- Add a description field to the update page and bind it the page model
- Modify the list view item template to display the age of the pet

8.7 Solutions

- Update the scrapbookUpdate-page.xml

```
<TextField text="{{ description }}" hint="Enter description..."/>
```

- Update the listView template to include the calcAge function:

```
<ListView.itemTemplate>
```

```
    <StackLayout>
```

```
<Label text="{{ title, title + ' Scrapbook Page ' + '(' + calcAge(year, month, day) + ')  
years old'}}" />  
</StackLayout>  
</ListView.itemTemplate>
```

9

Native hardware

This chapter covers

- How to load and save app data to a file
- How to use the device's camera to take and save pictures
- How to use GPS and WiFi signals to determine your location

In the first eight chapters of the book, you learned the basics of NativeScript: how it works, how apps are organized, how to create pages, how to add various UI elements to these pages, and how to manage data-entry using data binding. Through these topics, we laid the foundation of your NativeScript knowledge. In fact, you'll use that knowledge in every app you build. Even though these topics are foundation, they've focused on showing information on the screen of a mobile device, while leaving out an important aspect of mobile app development: interacting with native mobile device hardware.

On the surface, you may think using native mobile hardware components is difficult. Think about the number of Android and iOS devices that are on the market. Now think about the variety between devices: most devices are truly different: cameras, GPS modules, NFC (or no NFC), and even different Bluetooth specifications. Each device comes with a different set of capabilities, and with each, the potential for slight modifications to the underlying APIs. Because of these differences, it *can* be difficult to write code once and have it work across all the platforms. But, don't worry. NativeScript approaches this problem by abstracting commonalities between platforms and exposing a common API within the core modules.

In this chapter, you will learn about the core module APIs you can use to interact with the most common native hardware components of mobile devices: the file system, GPS, and camera. We'll start by updating the Pet Scrapbook app to use the file system of a device, so users of the app can save the scrapbook pages that they make. After this change, we'll continue to update the Pet Scrapbook to take and save pictures. Finally, you'll learn how to use the GPS to capture the location where your picture was taken.

9.1 The file system module

In the last chapter, you learned how to create more dynamic UIs with data-binding. We also reorganized the Pet Scrapbook to create parent and child views so the app was better-organized. While we were reorganizing the app, you may have noticed that all the pet data entered was deleted each time the app loaded. That's because we never saved the data: it was used only when the app was running, and removed as soon as the app was closed. Truthfully, we can't imagine a scrapbook app that deletes its data every time it's closed. In fact, not being able to save scrapbook pages makes the app unusable. Let's fix it.

Take a minute and think of a few ways you might solve this problem. Feel free to pull from any experience writing desktop, web, or mobile apps. Here's a few approaches we thought of:

5. Store the data in an external database, perhaps accessed via a web service. Each time the app starts, we could query the web service, request the saved data, and display it in the app. With each page we add or update, we could save the data back to the database using the same web service.
6. Store the scrapbook data locally (in a file or database), reading and updating the contents of the file as needed.

Both approaches would work just fine for the Pet Scrapbook, but this book is about working with NativeScript and mobile device hardware. So, let's look closer at the second approach and storing scrapbook data in a device's file system.

9.1.1 Using the file system module

To store data in a device's file system, you use the *file system module*.

DEFINITION The file system module is NativeScript core module. This module allows you to interact with the native file system of the device to find, retrieve, and store files and to interact with files in text or binary format.

NOTE If your app is deleted from the device, all the data stored on the device using the file system module will also be removed.

In the Pet Scrapbook, we want to store pet information entered into each page so we don't lose it when the app reloads. Using the file system module is straightforward, but let's review the basics of interacting with the file system before we apply in our app. Listing 9.1 shows how to store data in the file system, formatted as a JSON string.

Listing 9.1 Storing and retrieving a text file in NativeScript

```
var fileSystemModule = require("file-system"); // #A

exports.onLoaded = function() {
  var fileName = "myFile.json";
  var file = fileSystemModule.knownFolders.documents().getFile(fileName); // #B
  var data = {name: "Brosteins", type: "filesystemexample"};
  var jsonDataToWrite = JSON.stringify(data);
```

```
file.writeText(jsonDataToWrite); // #C
console.log("Wrote to the file: " + jsonDataToWrite);

var jsonDataRead = file.readTextSync(); // #D
console.log("Read from the file: " + jsonDataRead);

file.remove(); // #E
};

#A To use the file system module, you need to import it
#B Use the documents folder to store offline files that your app needs
#C Use the reference to a file to write data to the file system
#D Use the reference to the file to read the data. Data can be read synchronously or asynchronously
#E Delete the file
```

As you can see in listing 9.1, the file system module is used to access the file system of a device. When accessing the file system of a device, we are able to read and write files to the device in specific locations that our app has access to using the `readTextSync()` and `writeText()` functions. Using these functions, listing 9.1 references a file named `myFile.json`, then writes and read following the JSON object:

```
{name: "Brosteins", type: "filesystemexample"}
```

TIP Why JSON? JSON is easy to read, easy to write, and is the de facto text-based format for universal data interchange via JavaScript.

You'll also notice that listing 9.1 references a property on the file system module named `knownFolders`. Each NativeScript app has common folders that you can access using the file system module; these common folders are called *known folders*. Figure 9.1 shows the two different known folders that every NativeScript app can access: `documents` and `temp`.

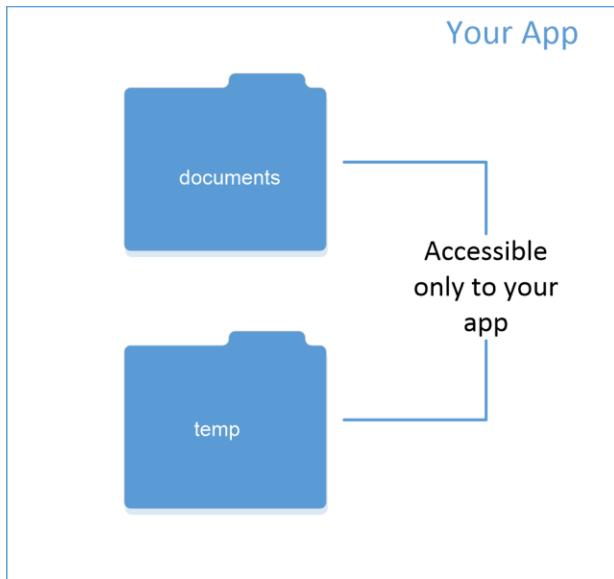


Figure 9.1 The known folders accessible to each NativeScript app.

Both known folders are *private*, meaning that they are only accessible to your app. The temp folder is generally used for caching or storing temporary data for your app (such as a webpage that we may be showing to the user in a pop up window), and the documents folder is for more permanent storage in your app. Permanent data for our app would be any data that we would want to maintain between user sessions, such as user-entered data, images that a user takes with the camera, or the saved state of a game.

HANDLING BINARY DATA

You just learned how to read and write text data to the file system, but you can also write binary data.

TIP You can write a binary data to a file by using the `writeSync()` function.

Even though we'll be working with text data in this chapter mostly, working with files in a binary format can come in handy when working with images or videos.

9.1.2 Integrating the file system module into the Pet Scrapbook

Now that you've learned the basics of the file system module, let's jump back into the Pet Scrapbook app and update it so it persists the data to the file system.

In chapter 8, we you'll recall that we passed the entire scrapbook model to the update page when a new scrapbook page was added or an existing scrapbook page list item was tapped (listing 9.2).

Listing 9.2 The `views\scrapbook-page.js` handling the tap event of an action item

```

exports.onAddTap = function(args) {
    var page = args.object;
    var scrapbook = page.bindingContext; //#A

    scrapbook.pages.push(new scrapbookPageModel());

    frame.topmost().navigate({
        moduleName: "views/scrapbookUpdate-page",
        context: { model: scrapbook, index: scrapbook.pages.length - 1 } //#A
    });
};

exports.onItemTap = function(args) {
    var page = args.object;
    var scrapbook = page.bindingContext; //#A

    frame.topmost().navigate({
        moduleName: "views/scrapbookUpdate-page",
        context: { model: scrapbook, index: args.index } //#A
    });
};

#A The scrapbook is an array of all scrapbook pages

```

Listing 9.2, recaps how data flows in the app at the end of chapter 8. The entire collection of scrapbook pages (the scrapbook variable) and the index to update is passed to the update page.

NOTE Passing the entire collection of scrapbook pages around the app may have seemed a bit confusing, especially because update page is only concerned about a single scrapbook page. We chose this approach in chapter 8 for two reasons: it helped to demonstrate different data and page-binding techniques in NativeScript and it was a simple way to get all the data that each page needed.

Now that we've learned about the file system module, we don't need to pass the entire collection of scrapbook pages around the app: we can retrieve and save individual pages as needed.

Before we dive right in to the code, check out figure 9.2, which describes how the Pet Scrapbook will change after we integrate the file system module.

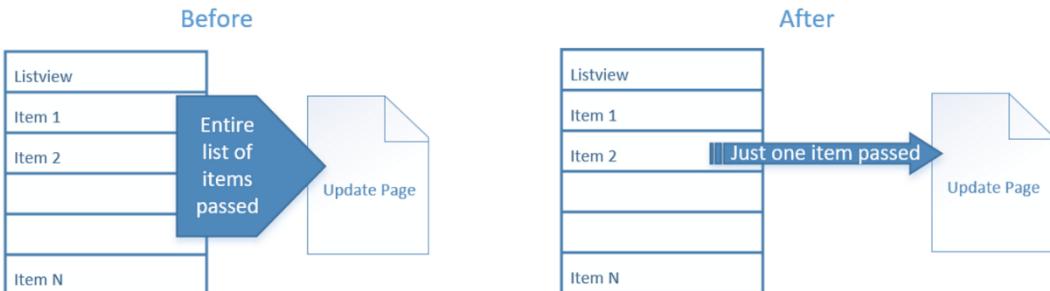


Figure 9.2 After integration of the file system to the Pet Scrapbook, single page objects instead of the entire list will be passed to the update page.

After the file system module is integrated, we'll store each page to the file system of the device. This will allow us to pass a single item to the update page. After the item has been updated, it will be saved back to the file system. Upon navigating back to the list of pages, we'll reload all the items from the file system. Figure 9.3 shows a breakdown of the responsibilities of these two pages.

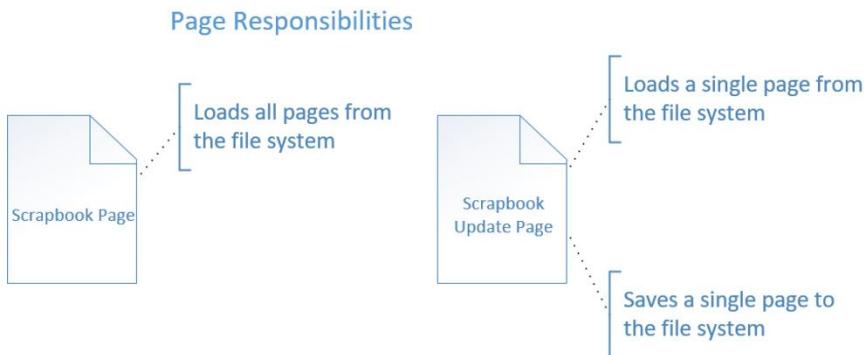


Figure 9.3 The responsibilities of the different pages of the Pet Scrapbook app.

As shown in figure 9.3, each page of the Pet Scrapbook app will have a unique purpose that matches the master/detail pattern described in chapter 8. The master page (scrapbook page) is responsible for displaying a list of items. The detail page (scrapbook update page) is responsible for updating the fields for a single item that is selectable in the master page.

Ok. Now that you understand where we're headed, let's get started with some code. We'll be refactoring the app with 5 steps:

- Step 1: Add a file system service
- Step 2: Add a unique identifier to a scrapbook page
- Step 3: Refactor the scrapbook list page to use the file system service
- Step 4: Refactor the navigation context passed to the update page
- Step 5: Refactor the update page to save new (and updated) pages

STEP 1: ADD A FILE SYSTEM SERVICE

The master and detail pages of the app will be sharing some functionality of the file system (reading and writing JSON data), so we think this is a great opportunity to break out some of that logic into a reusable code module that can be referenced on multiple pages. We like to refer to reusable modules as *service classes* or *service modules*.

DEFINITION A service class/module is a collection of reusable code that can be shared throughout an application to perform a specific group of related functionality. Service classes/modules typically create an internal API or intermediate layer of functionality in your code and sit between the front-end UI layer

of your application and data or file system access layers. Service classes/modules generally contain business logic.

Our service class will handle all the access to the file system, so let's call it the *file system service*. Add a new folder to the Pet Scrapbook called *data* (to house all of the data-related service classes) and add a file to it called *fileSystemService.js*.

TIP We have added the file system service to folder named *data*. This convention helps to keep our code organized by storing files that are associated with data access or data persistence in the same place. Keep this in mind as you create more components and features in apps that you are working on to help keep your code more organized and maintainable.

Figure 9.4 shows the resulting file structure of the Pet Scrapbook app.

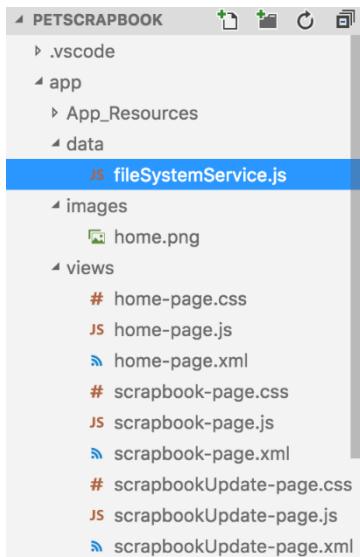


Figure 9.4 The file structure of the Pet Scrapbook app after adding the file system service.

After you have created the file system service, add the content of listing 9.3 to the file.

NOTE Listing 9.3 uses a JavaScript feature called *prototypical inheritance*. If you haven't run across prototypical inheritance before, it's not that bad. Just think of it as a way of defining different functions for the file system service. We prefer to use this approach because it defines a function for each instance of the file system service that we instantiate in code. To learn more about prototypical inheritance in

JavaScript you can visit: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain.

Listing 9.3 The `fileSystemService.js` file that will be used for file system interactions

```
var fileSystem = require("file-system"); // #A

var fileSystemService = function () {
    this.file = fileSystem.knownFolders.documents().getFile("scrapbook.json");
};

fileSystemService.prototype.getPages = function () { // #B
    var pages = [];

    if (this.file.readTextSync().length !== 0) {
        pages = JSON.parse(this.file.readTextSync()); // #C
    }

    return pages;
}

fileSystemService.prototype.savePage = function (scrapbookPage) { // #D
    var pages = this.getPages();

    var index = pages.findIndex(function (element) { // #E
        return element.id === scrapbookPage.id;
    });

    if (index !== -1) {
        pages[index] = {
            id: scrapbookPage.id,
            title: scrapbookPage.title,
            gender: scrapbookPage.gender,
            year: scrapbookPage.year,
            month: scrapbookPage.month,
            day: scrapbookPage.day
        };
    } else {
        pages.push({
            id: scrapbookPage.id,
            title: scrapbookPage.title,
            gender: scrapbookPage.gender,
            year: scrapbookPage.year,
            month: scrapbookPage.month,
            day: scrapbookPage.day
        });
    }
}

var json = JSON.stringify(pages); // #F
this.file.writeText(json); // #F
};

exports.fileSystemService = new fileSystemService();
#A Import the file system module
#B Method to retrieve the pages from the file system
```

```
#C Parse the JSON into objects
#D Method to save pages to the file system
#E Determine if the page already exists so we can update it
#F Convert objects to JSON and save to the file system
```

Listing 9.3 has a lot of code, so let's dissect it. We are creating a reusable component (much like the core modules in NativeScript) so that we can easily save and retrieve the scrapbook data from the file system. The file system service exposes two functions to: `getPages()` and `savePage()`.

The `getPages()` function reads scrapbook data from the file system and parses it into an array of scrapbook pages. We'll use the parsed array later to create an observable array of scrapbook pages for the scrapbook list view to display.

The `savePage()` function saves a single scrapbook page to the file system, and contains rudimentary business logic to check if the page already exists to determine if the page is updated or added (figure 9.5).

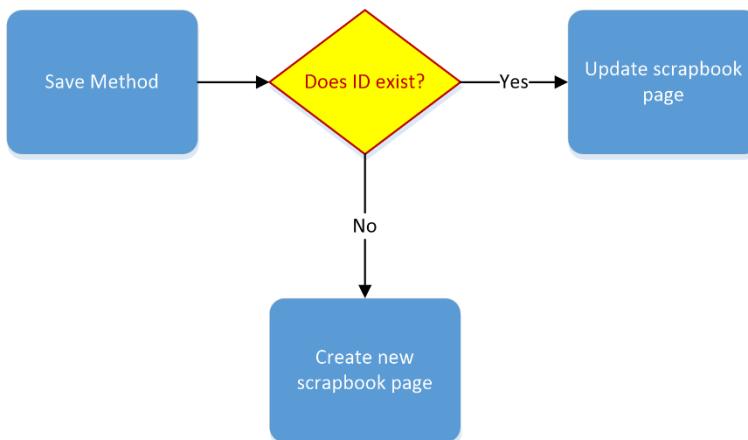


Figure 9.5 The decision process the file system service performs when a scrapbook page is saved.

Looking a little closer at the `savePage()` function, each scrapbook page has an `id` property that is used as a unique identifier. This makes it easy to tell if a scrapbook page is new or existing.

STEP 2: ADD A UNIQUE IDENTIFIER TO A SCRAPBOOK PAGE

The file system service we just created expects a scrapbook page to have unique identifier, so we'll have to make sure every scrapbook page created (and updated) has an `id`. The `scrapbookPageModel()` function on the scrapbook page is already used to create new scrapbook pages, so we just need to account for the `id` property in that function. Listing 9.4 shows the changes we made.

Listing 9.4 Updating `scrapbook.js` so the scrapbook page model has an `id` property

```
function scrapbookPageModel(id) { // #A
    var model = new observable.Observable(); // #A
```

```

model.id = id; // #A

model.genders = ["Female", "Male", "Other"];
model.calcAge = function(year, month, day){
    var date = new Date(year, month, day);
    var now = Date.now();
    var diff = Math.abs(now - date) / 1000 / 31536000;

    return diff.toFixed(1);
};

return model;
}

#A Force others to supply an id to create a scrapbook page

```

STEP 3: REFACTOR THE SCRAPBOOK LIST PAGE TO USE THE FILE SYSTEM SERVICE

With the file system service added, let's start using it on the scrapbook list page. At the end of chapter 8, the list view on the scrapbook list page was bound to the pages property of an observable. Listing 9.5 shows how the data was loaded into the observable.

Listing 9.5 Loading the list of scrapbook pages into an observable at the end of chapter 8

```

exports.onLoaded = function(args) {
    var page = args.object;
    var scrapbook;

    if(page.navigationContext != null) {
        scrapbook = page.navigationContext.model; // #A
    }
    else {
        scrapbook = new observable.fromObject({
            pages: new observableArray.ObservableArray(new scrapbookPageModel()) // #A
        });
    }

    page.bindingContext = scrapbook;
};

#A Scrapbook pages are loaded from the update page passing them back, or an empty array

```

Previously, scrapbook pages were loaded by tapping into the navigation context passed back from the update page (when we were adding/updating a page) or by creating a new observable array. Now that we've added the file system service, loading the list of scrapbook pages becomes much easier (listing 9.6).

Listing 9.6 The scrapbook.js with an updated onLoaded method to load scrapbook pages from the file system service

```

exports.onLoaded = function(args) {
    var page = args.object;
    var scrapbook = new observable.fromObject({
        pages: new observableArray.ObservableArray()
    });
    var pages = fileSystemService.fileSystemService.getPages(); // #A

```

```
if (pages.length !== 0) {
    pages.forEach(function (item) {
        var model = new scrapbookPageModel(item.id); // #B

        model.title = item.title;
        model.gender = item.gender;
        model.year = item.year;
        model.month = item.month;
        model.day = item.day;

        scrapbook.pages.push(model);
    });
}

page.bindingContext = scrapbook;
};

#A Load array of pages from the file system
#B Create an observable object and set the properties of the scrapbook page
```

When the page loads, the updated code created an observable array to hold scrapbook pages (see the `pages` property of the `scrapbook` object). After retrieving the saved scrapbook pages with the file system service, we check if any pages have been saved. For each saved page, we create a new scrapbook page model with the `scrapbookPageModel()` function and populate its fields. Finally, the page's binding context is set to the `scrapbook` object.

The beauty of this updated code is its simplicity: every time the page loads, it gets the data from the file system. This approach works well, assuming the update page saves new (and updated) pages to the file system. Let's make sure the update page does just that next.

STEP 4: REFACTOR THE NAVIGATION CONTEXT PASSED TO THE UPDATE PAGE

Now that we have integrated the file system service to load data from the file system of the device, we no longer need to pass the entire scrapbook around (because it is being persisted to the file system). All we really need to pass to the update page via the navigation binding context is a single scrapbook page. Figure 9.6 compares the navigation binding context being passed to the scrapbook update page before and after we integrate the file system into our app.

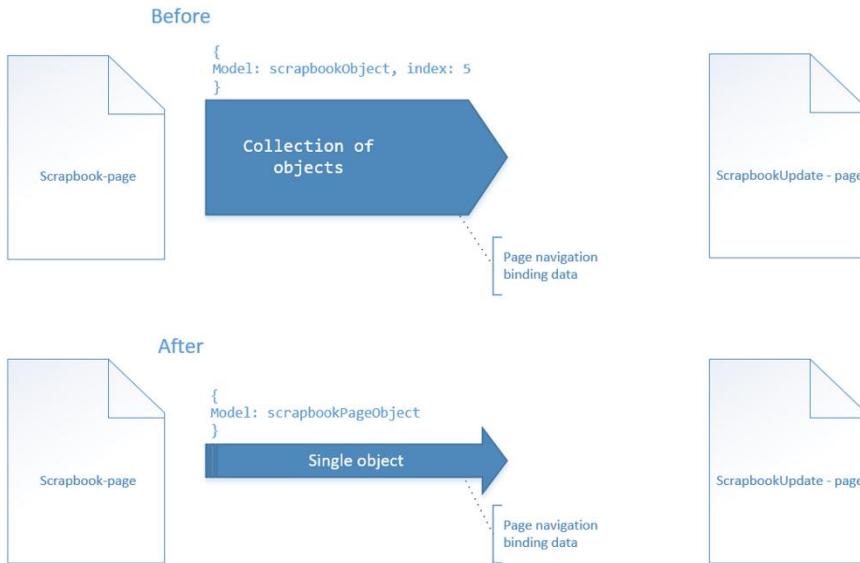


Figure 9.6 The before and after of the navigation binding data that is sent to the scrapbook update page when it is navigated to.

Let's check out the code changes we need to make to alter the navigation context being passed to the update page (listing 9.7).

Listing 9.7 Updated navigation binding context passed to the update page in scrapbook-page.js

```

exports.onAddTap = function(args) {
    var page = args.object;
    var scrapbook = page.bindingContext;

    frame.topmost().navigate({
        moduleName: "views/scrapbookUpdate-page",
        context: { model: new scrapbookPageModel(scrapbook.pages.length) } // #A
    });
};

exports.onItemTap = function(args) {
    var page = args.object;
    var scrapbook = page.bindingContext;

    frame.topmost().navigate({
        moduleName: "views/scrapbookUpdate-page",
        context: { model: scrapbook.pages.getItem(args.index) } // #B
    });
};

#A Pass a new scrapbook page model to the scrapbook update page
#B Pass the scrapbook page tapped to the scrapbook update page

```

To send a single scrapbook page object to the update page, we need to make changes to the add button tap event handler (the `onAddTap()` function) and the list view's item tap event handler (the `onItemTap()` function). The context passed to the update page from each function was adjusted to send a single scrapbook page object, but they were changed in different ways. For new pages (the `onItemTap()` function), we create a new scrapbook page object using the `scrapbookPageModel()` function. For existing pages, we lookup the item tapped and pass it directly.

TIP You may remember that we changed the `scrapbookPageModel()` function back in step 2 to require a unique identifier. Generating unique numbers can be difficult, but we used a cool JavaScript trick to generate our unique number. We used the length of the `pages` observable array as our unique identifier. When there's no items in the array, the length is 0, giving us a unique identifier of 0. When the page assigned with 0 is added to the observable array it will also be at index 0. This also means the next page added will have a unique id of 1 and an index of 1 in the observable array. Pretty cool!

The changes made to the scrapbook list page made the page simpler and easier to understand, but that's only half of the code. We also need to change the update page to account for a single object being passed via the navigation context. Listing 9.8 shows the change to the update page's loaded event handler.

Listing 9.8 Update loaded event handler supporting a single scrapbook page object in `scrapbookUpdate-page.js`

```
exports.onLoaded = function(args) {
    var page = args.object;
    var scrapbookPage = page.navigationContext.model;

    page.bindingContext = scrapbookPage;
};
```

STEP 5: REFACTOR THE UPDATE PAGE TO SAVE NEW (AND UPDATED) PAGES

The final step in our changes is to change the update page to save the update scrapbook page to the file system instead of passing the updated page back to the scrapbook list page via a navigation context. Listing 9.9 shows the updated done button tap handler code that saves the scrapbook page with the file system service.

Listing 9.9 Updated done button tap handler in `scrapbookUpdate-page.js`

```
var fileSystemService = require("~/data/fileSystemService");

exports.onDoneTap = function(args) {
    var page = args.object;
    var scrapbookPage = page.bindingContext;

    fileSystemService.fileSystemService.savePage(scrapbookPage); // #A

    frame.topmost().navigate({ // #B
        moduleName: "views/scrapbook-page" // #B
    });
};
```

```
}); // #B
};

#A Save the page using the file system service
#B Navigate back to the main page without passing a navigation context
```

And, we're finished. If you run the Pet Scrapbook app, you will notice that your scrapbook pages persist if you exit the app! There's not much to show visually (because we didn't touch the UI), but this change makes the app much more usable.

Now that you've learned how to use the file system, it is time to continue adding features to the scrapbook app and working with more hardware in NativeScript.

9.2 Camera

A foundation of modern smart phones is the ability to take, save, and share photos. But, so far, the Pet Scrapbook hasn't used this feature. In fact, we'd argue the Pet Scrapbook isn't much of a scrapbook because there aren't any photos! Let's change that by allowing users to use a device's camera to take photos (or select an existing photo from their photo album) and add them directly to a scrapbook page.

Before we get started, you'll need to add the `nativescript-camera` npm package to your app. Open a command line, navigate to your app's root folder, and run the following command:

```
npm install nativescript-camera --save
```

WARNING NativeScript originally shipped with a core module named `camera`, which provided the same capabilities as the `nativescript-camera` module. In October 2016, the `camera` core module was deprecated and moved to its own npm package. The move made sense, because it allowed for a faster development cycle of the package without having to wait for the core of NativeScript to be updated. As of January 2017, the `camera` core module still exists as part of NativeScript. We expect it will be removed from the core modules soon, so you shouldn't use it.

9.2.1 Taking photos

Taking photos on a mobile device is so ubiquitous that it requires no introduction, but how mobile devices take photos (and the options available while taking photos) varies widely across platforms and devices. Because of these differences, the `nativescript-camera` package approaches photos and the use of the camera in a minimalistic way: when you want to take a picture, call the `takePicture()` function. In turn, when the function is called, simply open the native device's camera UI and let the native device handle the rest. That's about as easy as it gets, so let's get started.

To integrate the camera and photos into the Pet Scrapbook app, we'll start by adding a button to the update page. When a user taps the button, we'll call the `takePicture()` function from the `nativescript-camera` module. After this step, we'll add the photo to the scrapbook page model and show it on the page. Let's start with the UI and add a button and image element to the update page (listing 9.10).

Listing 9.10 The scrapbookUpdate-page.xml with a camera button and image added

```
<Page loaded="onLoaded">
<StackLayout>
```

```

<Label text="{{ title, title + ' Scrapbook Page' }}" />
<TextField class="header" text="{{ title }}" hint="Enter title..." />

<Label text="{{ 'Age: ' + calcAge(year, month, day) + ' years old'}}" />
<DatePicker year="{{ year }}" month="{{ month }}" day="{{ day }}" />

<ListPicker items="{{ genders }}" selectedIndex="{{ gender }}" />

<Image src="{{ image }}" stretch="none" /> // #A

<Button tap="onAddImageTap" text="Add Image" /> // #B
<Button tap="onDoneTap" text="Done"/>
</StackLayout>
</Page>
#A The image's source is data-bound to the image property of the scrapbook page
#B Tapping the Add Image button will open the native camera UI to take or select a photo

```

The XML for the update page doesn't require much work because all we added was the image element and a button to open the native camera UI. One item to note is that the source property of the image element is data-bound to the image property of the scrapbook page's binding context. We haven't explicitly added this property to our scrapbook model yet, but don't worry: we will take care of that in just a bit.

NOTE Remember that the image element defaults to stretch the image to fill the screen. For now, we'll set the image to not stretch to maintain the aspect ratio, but in the next chapter, we'll revisit styling and make the page look more professional.

When the user taps the Add Image button, we want to launch the native camera UI, allow them to take or select a picture. Listing 9.11 shows the code added to handle the button's tap event handler.

Listing 9.11 Add image button tap event handler added to scrapbookUpdate-page.js

```

var camera = require("nativescript-camera"); // #A
var image = require("image-source"); // #A

exports.onAddImageTap = function (args) {
    var page = args.object;
    var scrapbookPage = page.bindingContext;

    camera.requestPermissions(); // #B
    camera
        .takePicture() // #C
        .then(function (picture) { // #D
            image.fromAsset(picture).then(function (imageSource) { // #E
                scrapbookPage.set("image", imageSource); // #E
            });
        });
}

#A Importing the nativescript-camera module and image-source module is the same as any core module
#B To use the camera you need to request permission
#C takePicture() returns a promise
#D When the promise resolves, the then() function is called, passing the picture
#E Create an image source object to bind to the view

```

As we've mentioned, using the camera module is straightforward, but there are a few items we want to point out. Before taking a picture with the camera, you'll need to ask the mobile device if it's okay to use the camera by calling the `requestPermissions()` function. When this method is called, the mobile device will prompt the user to grant access to use the camera. Figure 9.7 shows the different messages on Android and iOS.

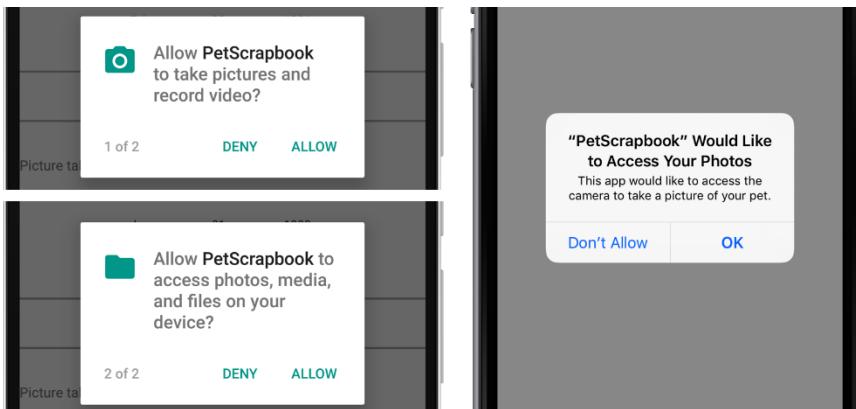


Figure 9.7 Android (left) and iOS (right) requesting permission to access the camera and photos.

TIP You must request permission to access the camera only once, but it's a good practice to request permission before taking a picture, even if you believe you've asked already. Multiple calls to the `requestPermissions()` function are ignored once you've been granted access to use the camera.

After requesting permission to the camera, you can take a picture with the `takePicture()` function. When called, the function returns a JavaScript *promise* after opening the native device's camera UI. The promise will get resolved when the native camera UI returns a photo that was taken or selected from the device's photo gallery. The resolved promise contains a reference to the picture. Conveniently, the returned picture is exactly what the image element needs to data-bind to, so we can add it directly to the scrapbook page model and the data-binding will take care of the rest.

DEFINITION A promise is a JavaScript way of doing asynchronous code execution. We're not going to cover how promises work, but you can learn more about promises at <https://developers.google.com/web/fundamentals/getting-started/primers/promises>.

Let's look at the scrapbook after making these changes. Figure 9.9 shows Android and iOS camera UIs shown after tapping the Add Image button.

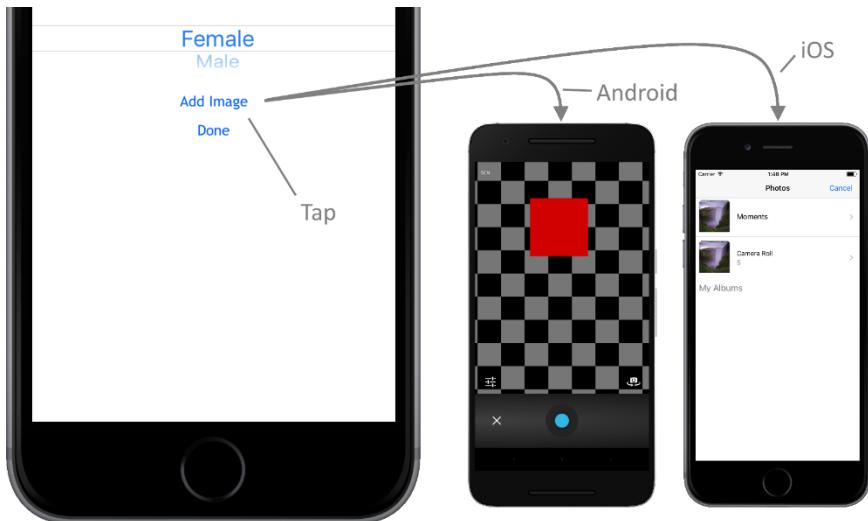
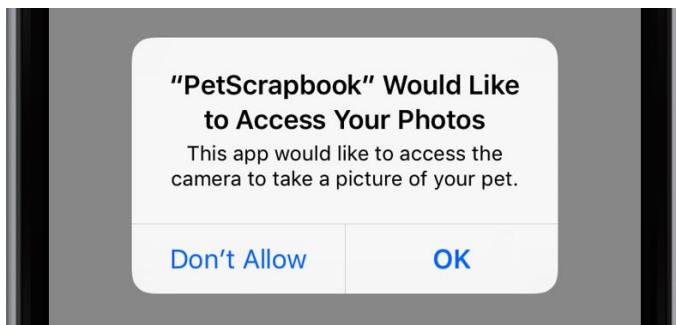


Figure 9.9 The launching of the camera in Android (middle) and iOS (right) when the add image button is tapped.

Differences with the camera in Android and iOS

The Android emulator simulates a camera, but the iOS simulator does not. If you are running your app using the iOS simulator, you will be able to choose only an image that is in the photo library of the device; the resulting image object that is returned by the camera module is the same whether an image is chosen from the library or taken by the camera.

In iOS, the first time you launch the camera from a new app, the user will be prompted with a security dialog as shown in the following figure.



The security dialog that a user is prompted with in iOS when an app tries to access the camera the first time.

The description text of the security dialog can be customized for iOS by updating the `NSPhotoLibraryUsageDescription` key/value pair inside of the `Info.plist` file.:.

```
<key>NSPhotoLibraryUsageDescription</key>  
<string>This app would like to access the camera to take a picture of  
your pet.</string>
```

The info.plist file is an information property list file. This file is used by iOS apps to provide metadata to iOS. iOS understands the structure of the info.plist file (a system key/value pair collection) and is able to access the file at runtime. The info.plist file is located within the platform-specific folder files discussed in chapter three at *app/App_Resources/iOS/Info.plist*. For more information about the info.plist file, you can review official apple documentation at <https://developer.apple.com/library/content/documentation/General/Reference/InfoPlistKeyReference/Introduction/Introduction.html>

After a user selects an image or takes a picture with the camera, the camera module automatically returns the user to the page that they were on, as seen in figure 9.10.

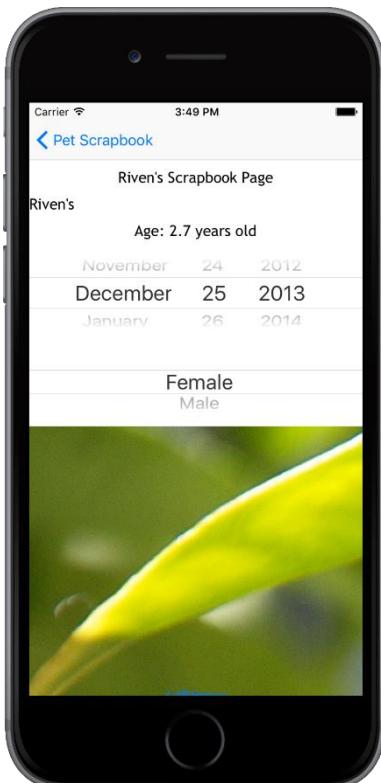


Figure 9.10 The scrapbook page after taking a picture.

You may have noticed that the image you selected or captured is rather large. Most device cameras have a very high megapixel count, which could result in taking images that consume a lot of space on a device and are physically large as well. Luckily, the camera module allows you to adjust the size of the picture by passing in a parameter to the `takePicture()` function (listing 9.12).

Listing 9.12 Passing parameters to the `takePicture()` function on the update page to control the image size

```
exports.onAddImageTap = function (args) {
    var page = args.object;
    var scrapbookPage = page.bindingContext;

    camera.requestPermissions();
    camera
        .takePicture({
            width: 100, // #A
            height: 100, // #A
            keepAspectRatio: true}) // #B
        .then(function (picture) {
            image.fromAsset(picture).then(function (imageSource) {
                scrapbookPage.set("image", imageSource);
            });
        });
}
//A Set the height and width of the image we take with the camera
//B Tell the camera module to maintain the aspect ratio of the picture taken
```

The `takePicture()` function takes an optional JSON object as a parameter that accepts four optional properties: `width`, `height`, `keepAspectRatio`, and `saveToGallery`. Table 9.1 summarizes how each of these properties can be used.

Table 9.1 Optional properties and parameter values for the `takePicture()` function

Name	Description
width	The maximum (or desired) width of the picture (in device independent pixels).
height	The maximum (or desired) height of the picture (in device independent pixels).
keepAspectRatio	A true/false value indicating whether the image's original aspect ratio (or dimensions) should be enforced.
saveToGallery	A true/false value indicating if the photo should be saved to the mobile devices photo gallery. This is the "Photos" area on Android and the "Camera Roll" on iOS.

After making these changes, the picture taken appears smaller on our device (figure 9.11).

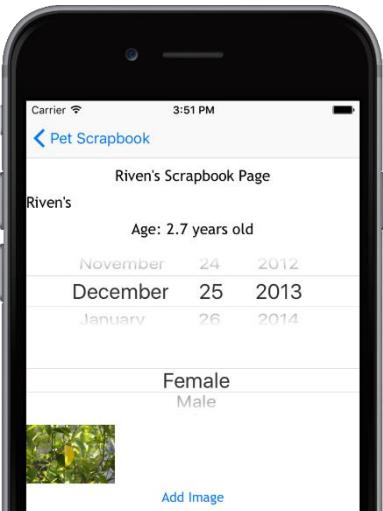


Figure 9.11 The scrapbook update page showing an image that has been taken at a lower resolution using the camera module.

WIDTH, HEIGHT, AND KEEP ASPECT RATIO EXPLAINED

You may have noticed that the picture taken in figure 9.11 isn't 100 x 100 pixels, even though we specified a width and height of 100 pixels. This is because we requested the app maintain the *aspect ratio* of the picture.

DEFINITION The aspect ratio of a picture is the relationship between the picture's width and height, expressed in the format #:#, read "# by #." For example, an image with a width of 300 pixels and height of 400 pixels is referred to as having an aspect ratio of 3:4, or read "three by four." Further, a 100 x 100 pixel image has an aspect ratio of 1:1.

By telling the app to retain an image's aspect ratio, the *nativescript-camera* package automatically readjusts the height and width of the picture so the image's aspect ratio is maintained, but is resized to the desired width or height. In figure 9.11, the image's aspect ratio is 3:2, so the image is resized to be 100 x 67 pixels.

9.2.2 Saving the image to the file system

Now that we have integrated the camera into the Pet Scrapbook, we'll need to provide a way for saving our scrapbook images to the file system. This means, we'll make a few changes to the file system service we created earlier in this chapter. But, before we start writing code, we need to decide on how we'll save the binary image data to a text file.

WARNING Hold on! Binary data? If you've never worked with images before, then you may not know that images are stored as binary data, not text data. This means you must store, read, and write the

image data as binary data or convert the binary image data to text data before writing it to the file system.

Earlier in this chapter, we mentioned that we'd be storing our scrapbook data in a text file, formatted as a JSON string. We also used the `readText()` and `writeText()` functions of the file system module, which read and write text-formatted data. This poses a problem because our image data is binary-formatted. There's a few ways we could tackle this problem, including converting the binary data to text data and saving each image as its own binary file on the file system. To keep things simple, let's convert the binary image data to text data using base64 encoding.

DEFINITION Base64 encoding is a common binary to string encoding scheme that will take a binary file and represent it as an ASCII string.

TIP Even though we've decided to keep things simple by using base64 encoding, converting images to base64-encoding can increase the image's size by up to 1.33x. As we add more and more images to the pet scrapbook, the size of the file we're using to store all the scrapbook data could get rather large. If we were planning to have hundreds of scrapbook pages, we may want to reconsider storing all the images as base64-encoded strings in a single file. Instead, we may store each image individually as a binary file, and store the name of that file as text data in the main data file. With this approach, we could load individual image files, as needed, instead of everything all at once.

Let's revisit the file system service from earlier in the chapter and add support for reading and saving images as base64-encoded strings. We'll do this by adding an additional property named `imageBase64` to our JSON file. Before writing the JSON file, we'll convert the binary image to a base64-encoded string and place the value in the `imageBase64` property. When the JSON file is read from the file system, we'll reverse the process, converting the base64-encoded string to a binary image. Figure 9.13 summarizes this process.

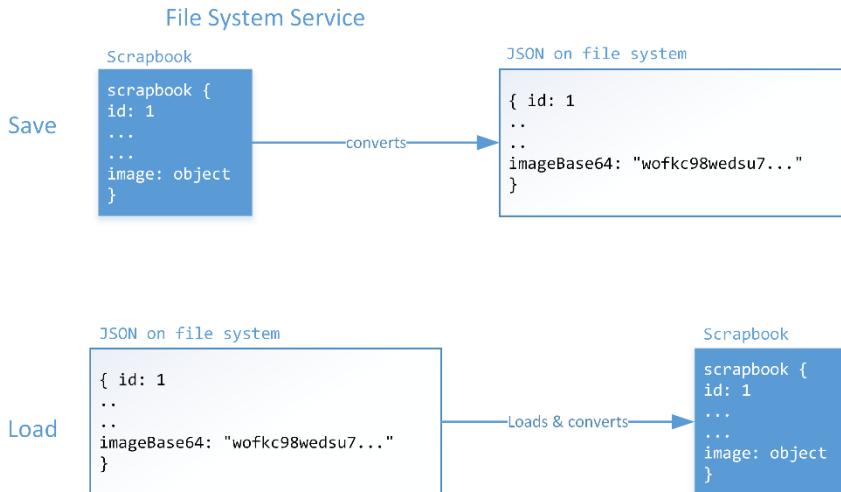


Figure 9.13 The file system service will convert binary image data to a base64-encoded string when saving to the file system. The process is reversed when data is read from the file system.

Luckily for us, NativeScript's image module already has implementations for converting images to and from base64 encoded strings. Listing 9.13 shows the updates to the file system service to support saving images to the file system.

Listing 9.13 The `data/fileSystemService.js` file updated to save and load images

```
var fileSystem = require("file-system");
var imageModule = require("image-source"); // #A

var fileSystemService = function () {
    this.file = fileSystem.knownFolders.documents().getFile("scrapbook.json");
};

fileSystemService.prototype.getPages = function () {
    var pages = [];

    if (this.file.readTextSync().length !== 0) {
        pages = JSON.parse(this.file.readTextSync());
    }

    pages.forEach(function (page) {
        if (page.imageBase64 != null) {
            page.image = imageModule.fromBase64(page.imageBase64); // #B
        }
    });
}

return pages;
};

fileSystemService.prototype.savePage = function (scrapbookPage) {
```

```
var pages = this.getPages();

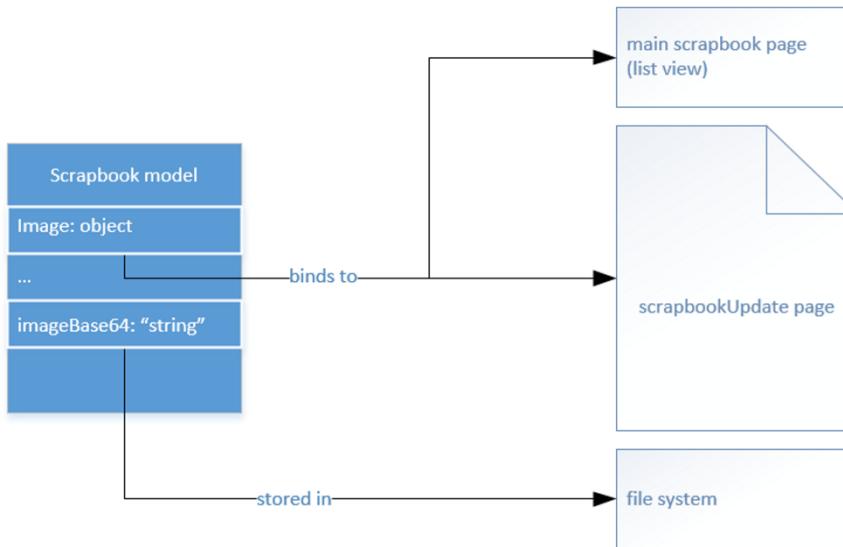
var index = pages.findIndex(function (element) {
    return element.id === scrapbookPage.id;
});

if (index !== -1) {
    pages[index] = {
        id: scrapbookPage.id,
        title: scrapbookPage.title,
        gender: scrapbookPage.gender,
        year: scrapbookPage.year,
        month: scrapbookPage.month,
        day: scrapbookPage.day,
        imageBase64: scrapbookPage.image != null ?
            scrapbookPage.image.toBase64String("png") : null // #C
    };
}
else {
    pages.push({
        id: scrapbookPage.id,
        title: scrapbookPage.title,
        gender: scrapbookPage.gender,
        year: scrapbookPage.year,
        month: scrapbookPage.month,
        day: scrapbookPage.day,
        imageBase64: scrapbookPage.image != null ?
            scrapbookPage.image.toBase64String("png") : null // #C
    });
}

var json = JSON.stringify(pages);
this.file.writeText(json);
};

exports.FileSystemService = new FileSystemService();
#A The image module has a base64 encoder/decoder
#B Convert each image string loaded to an image object
#C Convert the image into a base64 string to store with the scrapbook page
```

When we updated the file system service, we created a new property on the scrapbook page named *imageBase64*. As seen in figure 9.14, the *imageBase64* property is used only by the file system service. We also created the *image* property, which is used to bind to image elements on the two scrapbook pages.



Figured 9.14 The image and imageBase64 properties are used by the Pet Scrapbook app in different ways. The image object is used for data-binding, and the imageBase64 string is used to save the image to the file system.

9.2.3 Displaying the image

Now that we can store and retrieve the image object from the file system, the main scrapbook page, let's update the main scrapbook page to show a thumbnail image of our pet. Listing 9.14 and 9.15 show how to add an image property to the page's binding context and bind it to an image element in the list view.

Listing 9.14 The scrapbook-page.js `onLoaded()` function updated to load the image into the scrapbook model

```

exports.onLoaded = function(args) {
    var page = args.object;
    var scrapbook = new observable.fromObject({
        pages: new observableArray.ObservableArray() });
    var pages = fileSystemService.fileSystemService.getPages();

    if (pages.length !== 0) {
        pages.forEach(function (item) {
            var model = new scrapbookPageModel();

            model.id = item.id;
            model.title = item.title;
            model.gender = item.gender;
            model.year = item.year;
            model.month = item.month;
            model.day = item.day;
            model.image = item.image; // #A

```

```
        scrapbook.pages.push(model);
    });
}
else {
    scrapbook = new observable.fromObject({
        pages: new observableArray.ObservableArray()
    });
}

page.bindingContext = scrapbook;
};

#A Set the image property of the scrapbook model loaded from the file system
```

Listing 9.15 The scrapbook-page.xml binding to the scrapbook image

```
<Page loaded="onLoaded">
<Page.actionBar>
    <ActionBar title="Pet Scrapbook">
        <ActionItem tap="onAddTap" ios.position="right"
            text="Add" android.position="actionBar"/>
    </ActionBar>
</Page.actionBar>
<StackLayout>
    <ListView items="{{ pages }}" itemTap="onItemTap">
        <ListView.itemTemplate>
            <StackLayout orientation="horizontal" // #A
                <Image src="{{ image }}"/> // #B
                <Label text="{{ title, title + ' Scrapbook Page' }}"/>
            </StackLayout>
        </ListView.itemTemplate>
    </ListView>
</StackLayout>
</Page>
```

#A Create a horizontal layout for thumbnails
#B Bind the image to a UI image

By wrapping the label with a horizontal stack layout and adding an image, we created the look and feel of a thumbnail in the list view item template. Figure 9.15 shows the result, with the image and label displayed side-by-side.

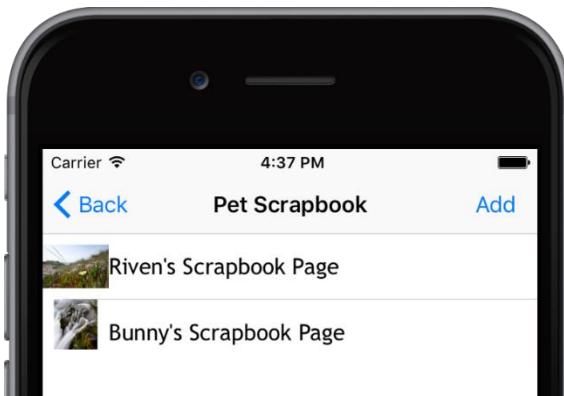


Figure 9.15 The scrapbook page updated to show the pet image like a thumbnail.

9.3 Using GPS and location services

The global positioning system (GPS) is another mobile device feature you'll use throughout the apps you create. GPS is often used in conjunction with other services, like maps to display your location, social sharing to tell others where you're visiting, and photos to identify where a photo was taken. We'll be integrating GPS into the Pet Scrapbook so we can capture the location where scrapbook images are taken.

Although people usually refer to finding their location as "using GPS," modern mobile devices use more than GPS to determine your location. Because of this distinction, you'll likely see location-related functionality of a mobile device referred to as "using *location services*."

DEFINITION Location services is group of mobile device capabilities that identifies a user's location.

Most mobile devices combine both GPS and WiFi signals to help determine your location to a high degree of accuracy.

Figure 9.16 shows how these two systems work together to produce location data that can be used in apps.

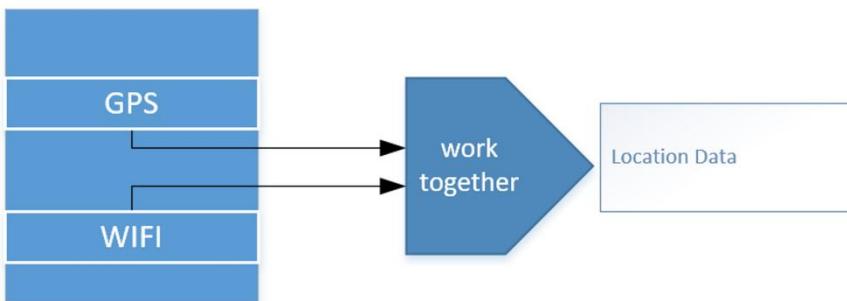


Figure 9.16 Location services combine GPS and WiFi to create location data.

Now that you know how mobile devices obtain your location with location services, let's add it to the Pet Scrapbook. We'll be using the *nativescript-geolocation* npm package to access location services. You'll recall from earlier in this chapter that you can add npm packages to your app via the command line: `npm install <npm package name> --save`. This holds true for the *nativescript-geolocation* package, but there's a second way to add packages to your app by using the NativeScript plugin system.

9.3.1 Plugins

NativeScript plugins are a fancy way of saying, an npm package specifically written for NativeScript apps. You've already seen (and used) a variety of plugins, but probably didn't realize it. For example, the NativeScript core modules and the *nativescript-camera* package are both plugins. In fact, there are over

400 plugins for NativeScript, ranging from custom UI controls, to hardware devices like the accelerometer, and even barcode scanners. Now, that's cool!

TIP The official NativeScript plugin site is <http://plugins.nativescript.org>. From there, you can browse through hundreds of plugins.

If the core modules and the camera package we added earlier in this chapter are plugins, you might be wondering if all npm packages are plugins? Not exactly. There are a few differentiating points for a regular npm package to be considered a NativeScript plugin, but truthfully, you don't need to know about these unless you're planning to write your own plugin. Writing your own plugins is a cool and interesting topic, but we're not going to cover it in this book.

NOTE You can find more information on plugins at <https://docs.nativescript.org/plugins/plugins> regarding the npm package structure. Most NativeScript plugins are named using the nativescript-pluginname convention and can be found by searching for NativeScript on <https://npmjs.org>.

9.3.2 Using the geolocation plugin

As we mentioned earlier, you can add any npm package to your app using the npm CLI, but NativeScript has a dedicated CLI option to do the same thing. Let's use the NativeScript CLI to add the geolocation plugin. Navigate to the root folder of the Pet Scrapbook and run the following CLI command:

```
tns plugin add nativescript-geolocation
```

After running the command, the *nativescript-geolocation* plugin is added to the app. If you look in the *node_modules* folder, you should see a new folder (figure 9.18).

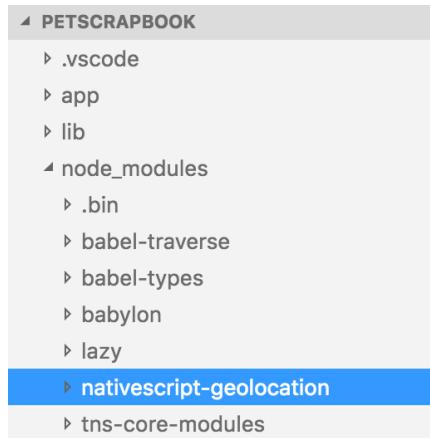


Figure 9.18 Adding the *nativescript-geolocation* plugin to the Pet Scrapbook adds a folder to the *node_modules* folder.

NOTE Don't forget that all plugins are npm packages, and that's why the geolocation plugin is added to the `node_modules` folder.

TIP You can also install plugins by using npm directly. For example, running `tns plugin add nativescript-geolocation` is the same as running `npm install nativescript-geolocation --save`.

Now that we've added the geolocation plugin, we can use it the same way we use any other NativeScript core module. Before we jump into the code, let's plan how we'll incorporate location services into the app. Figure 9.19 shows how we'll be modifying the app's behavior when the Add Image button is pressed.

NOTE Remember when we added the camera to the Pet Scrapbook and the user was prompted to allow the app to access the camera? Accessing location data from within an app requires us to prompt the user in a similar way. This isn't hard to do because it's done automatically, but we feel it's important to point out.

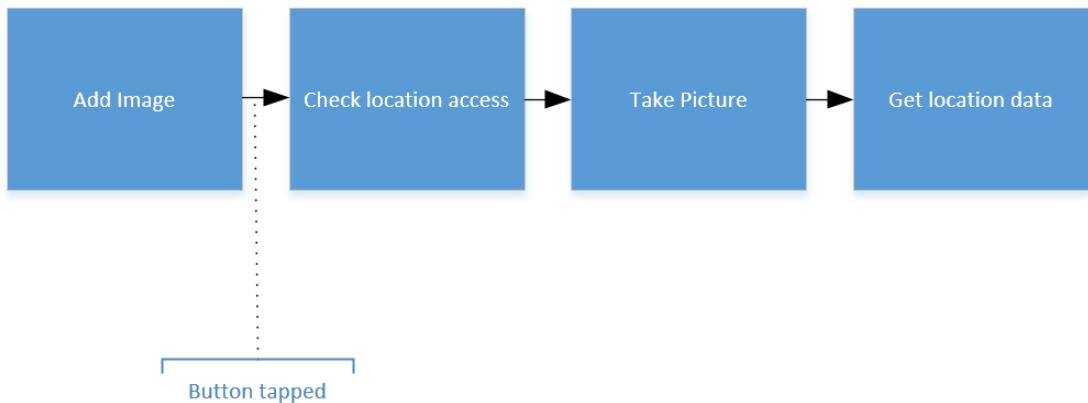


Figure 9.19 The workflow when the add image button is tapped.

With the updated workflow in mind, update the Add Image button tap event handler to incorporate a request to use location services and a call to get the current location (listing 9.16).

Listing 9.16 The scrapbookUpdate-page.js updated to get location data

```
var geolocation = require("nativescript-geolocation"); // #A

exports.onAddImageTap = function (args) {
    var page = args.object;
    var scrapbookPage = page.bindingContext;
```

```

if (!geolocation.isEnabled()) { // #B
    geolocation.enableLocationRequest(); // #B
}

camera
    .takePicture({ width: 100, height: 100, keepAspectRatio: true })
    .then(function (picture) {
        image.fromAsset(picture).then(function (imageSource) {
            scrapbookPage.set("image", imageSource);
        });
    });

    geolocation.getCurrentLocation().then(function (location) { // #C
        scrapbookPage.set("lat", location.latitude); // #D
        scrapbookPage.set("long", location.longitude); // #D
    });
};

#A Add the geolocation package reference to the top
#B You should check to see if location services is enabled before using it, and request that it be enabled
#C Getting the location data automatically prompts the user for permission
#D The returned location has latitude and longitude values

```

Before using the geolocation package to access location services on a mobile device, you should check to see if it's enabled. Most modern mobile devices disable location services by default because it can cause the device's battery to drain quickly. Listing 9.16 shows how the `isEnabled()` function of the geolocation plugin is used to check if location services is enable for the app. If it's not enabled, the `enableLocationRequest()` function is used to request the user enable location services.

NOTE The `isEnabled()` and `enableLocationRequest()` functions of the geolocation module are nice API abstractions, wrapping the native implementation of the necessary native API calls needed to ensure location services is enabled. Using these functions makes it simple to ensure you have access to location services. If the user doesn't enable location services, requests for location data will return nothing.

Once we've verified we have access to location services, the `getCurrentLocation()` function queries the mobile device's location services, returning your location. You'll notice that a JavaScript promise is actually returned, so you'll have to use the `then()` function syntax to retrieve the latitude and longitude.

Finally, let's add a label to the update page so we can see the location where the picture was taken (listing 9.17).

Listing 9.17 The `scrapbookUpdate-page.xml` showing the new location properties data bound

```

<Page loaded="onLoaded">
    <StackLayout>
        <Label text="{{ title, title + ' Scrapbook Page' }}" />
        <TextField class="header" text="{{ title }}" hint="Enter title..."/>
        <Label text="{{ 'Age: ' + calcAge(year, month, day) + ' years old'}}" />
        <DatePicker year="{{ year }}" month="{{ month }}" day="{{ day }}" />
        <ListPicker items="{{ genders }}" selectedIndex="{{ gender }}" />
        <Image src="{{ image }}" stretch="none" />

```

```
// #A
    <Label text="{{ (lat, long), 'Picture taken at ' + lat + ', ' + long }}" />
</Page>
```

#A Bind the latitude and longitude to the update page

Let's run the Pet Scrapbook and see our changes in action. Figure 9.20 shows the message displayed when the Add Image button is tapped.

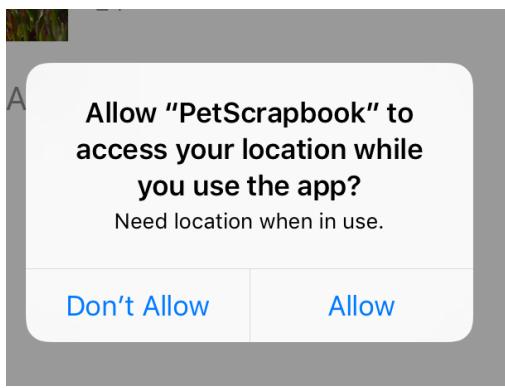


Figure 9.20 The security prompt that happens the first time that location data is accessed.

NOTE The security prompt to allow location access will occur only the first time that the app is launched. This behavior is managed by the Android and iOS operating system.

After allowing the Pet Scrapbook to access location data on the device and then taking a picture, you'll see that the UI is now updated with the latitude and longitude, as shown in figure 9.21. You may have also noticed in listing 9.14 we expanded upon the data binding expressions that we learned in the last chapter by creating an expression that has two parameters (`lat` and `long`). To use multiple parameters in a binding expression, you need to use parenthesis to group them before delimiting them with a comma from the expression.

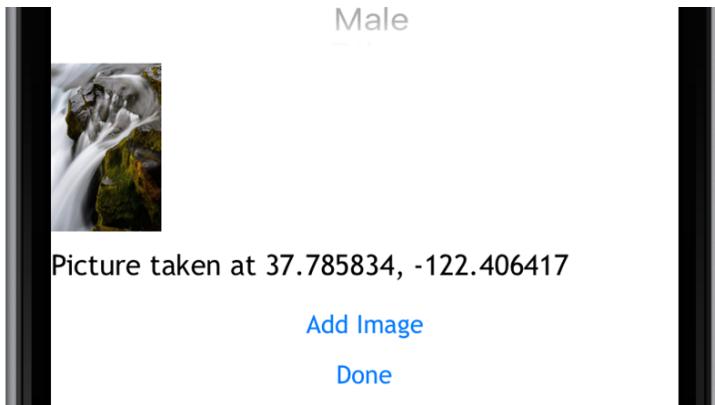


Figure 9.21 The scrapbook update page showing the data binding of location information.

9.4 Summary

In this chapter, you learned to do the following:

- Interact with device hardware
- Store and retrieve files on the file system
- Use the camera and GPS of a device
- Install and use a NativeScript plugin

9.5 Exercise

3. Add a timestamp to the scrapbook page when it is saved
7. Change the aspect ratio when taking a picture with the camera to force a 4x3 ratio

9.6 Solutions

4. Update the savePage method of the data\fileSystemService.js file

```
fileSystemService.prototype.savePage = function (scrapbookPage) {
    var pages = this.getPages();

    var index = pages.findIndex(function (element) {
        return element.id === scrapbookPage.id;
    });

    if (index !== -1) {
        pages[index] = {
            id: scrapbookPage.id,
            title: scrapbookPage.title,
            gender: scrapbookPage.gender,
            year: scrapbookPage.year,
            month: scrapbookPage.month,
```

```
        day: scrapbookPage.day,
        imageBase64: scrapbookPage.image != null ?
scrapbookPage.image.toBase64String("png") : null,
        timestamp: new Date().toString()
    );
}
else {
    pages.push({
        id: scrapbookPage.id,
        title: scrapbookPage.title,
        gender: scrapbookPage.gender,
        year: scrapbookPage.year,
        month: scrapbookPage.month,
        day: scrapbookPage.day,
        imageBase64: scrapbookPage.image != null ?
scrapbookPage.image.toBase64String("png") : null,
        timestamp: new Date().toString()
    });
}

var json = JSON.stringify(pages);
this.file.writeText(json);
};
```

2. Update the `camera.takePicture()` function call to the following:

```
camera.takePicture({ width: 400, height: 300, keepAspectRatio: false })
```

10

Creating professional UIs with themes

This chapter covers

- Styling your apps with NativeScript themes instead of custom CSS
- Creating consistent-looking data entry forms to improve user experience

Over the last several chapters you've learned about data binding and interacting with mobile device hardware by creating the Pet Scrapbook app. The Pet Scrapbook app is functional, but it doesn't feel like a well-designed app: it's a little clunky. We've neglected the app's visual design, the detail page is just hard to use, and the app is built for only phone-sized devices. It's time to fix these issues and make the app feel polished.

Over the next two chapters, you'll be learning how to turn functioning apps into more polished apps by using more advanced UI design techniques. What constitutes a more polished app is somewhat subjective, but there are solid UI design patterns that you can use to improve an app. In this chapter, we'll start a journey of refactoring and refinement of the Pet Scrapbook app by using NativeScript themes to style the UI. Let's get to it!

10.1 Themes

We may be stretching a bit, but bear with us, because we're taking a trip down nostalgia lane. If you've done a fair bit of web development, you might remember the good 'ole days—hand-coding UIs with CSS, margins, paddings, floats, clearing floats. It was glorious. Or not. Then there was Bootstrap. Structure your HTML markup in a certain way, add a few classes, and your apps started to look good (at least for us non-designers). Sure, everyone's website started to look the same, but heck, we'll take cookie-cutter styling over the horror of floats any day.

Ok, enough reminiscing. In all seriousness, designing great UIs is hard, even if you're a designer. And, we're not designers, so we like to use tools that make it easy to create good looking apps. That's where themes come in.

DEFINITION Themes are a collection of pre-built CSS style rules you can use to quickly style your apps. Think of themes as Bootstrap for NativeScript apps.

By following a few XML markup conventions and applying a CSS style, your apps can come to life. But let's be clear: just because you're using themes doesn't mean that you'll never write CSS for your app again. You'll still need CSS, but just not nearly as much.

10.1.1 Incorporating themes into your app

The first stop on our refactoring journey is to re-style our app with themes. We'll be updating each page by removing some of our custom CSS, and adding pre-defined CSS classes provided by NativeScript themes. We think the best way to learn something new is through practice, so let's get started.

As we've said earlier, themes are a collection of pre-built CSS class selectors. The theme collection is maintained as an npm package named *nativescript-theme-core*, and it's automatically included in your app if you created your app using NativeScript version 2.4.0 or above.

TIP If you don't know what version of NativeScript you used to scaffold your app initially, it's ok. Check the *package.json* file in the root of your app. If the *nativescript-theme-core* package is installed, you'll see it listed in the *dependencies* section. For example: "nativescript-theme-core": "[^]1.0.2".

If you don't have the *nativescript-theme-core* package installed, you can install it from the command line using *npm*:

```
npm install nativescript-theme-core --save
```

Figure 10.1 shows what you should see after adding the theme package to the app.

```
mikeb-macbook-pro:PetScrapbook mike$ npm install nativescript-theme-core --save
nativescript-theme-core@1.0.2 node_modules/nativescript-theme-core
mikeb-macbook-pro:PetScrapbook mike$ █
```

Figure 10.1 Console output after adding the *nativescript-theme-core* package to the Pet Scrapbook.

During the installation process, *npm* adds *nativescript-theme-core* to the *node_modules* folder, as seen in figure 10.2.

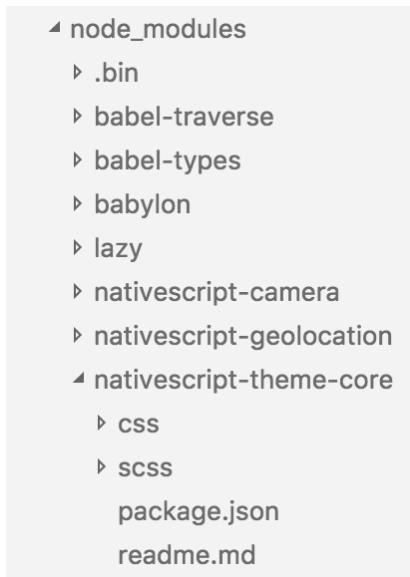


Figure 10.2 The Pet Scrapbook node_modules folder showing the added themes package.

Several color schemes are included in *nativescript-theme-core* – two core schemes: light and dark, and eleven other colors (blue, grey, orange, purple, and so on). All the color schemes provide the same capabilities, but colorize UI elements differently. Figure 10.3 shows the light (top) and dark (bottom) color schemes for Android (left) and iOS (right).

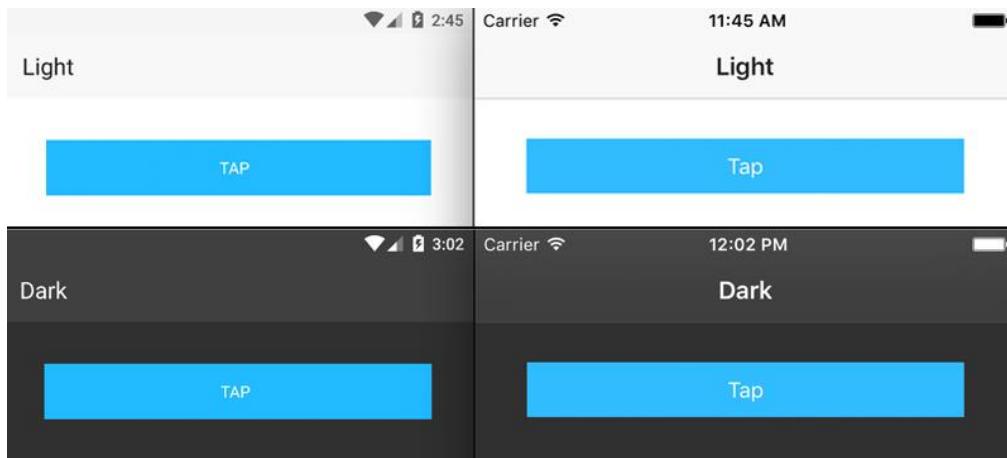


Fig 10.3 Light (top) and dark (bottom) color schemes applied to an app on Android (left) and iOS (right).

To use a color scheme, import the CSS file into your app at the top of the app.css file. To import the light theme, add this reference:

```
@import 'nativescript-theme-core/css/core.light.css';
```

TIP Switching color schemes is easy. If you'd like to see use the dark color scheme, remove the core.light.css import statement at the top of the app.css file and add `@import 'nativescript-theme-core/css/core.dark.css';`. Explore the css folder of the nativescript-theme-core npm package for other color schemes (blue, for example). To switch to the alternate color, import the color: `@import 'nativescript-theme-core/css/blue.css';`.

We're going to use the light color scheme in the Pet Scrapbook. Be sure to import it into the `app.css` file by adding the import statement at the top:

```
@import 'nativescript-theme-core/css/core.light.css';
```

10.2 Using text classes, alignment, and padding

We'll start using themes on the home page of the Pet Scrapbook. At the end of chapter 9, the home page had several custom CSS classes for the page header, footer, and labels (listing 10.1).

Listing 10.1 Custom CSS classes in home-page.css at the end of Chapter 9

```
.header {  
    font-size: 32px;  
}  
  
label {  
    text-align: center;  
    margin-top: 10px;  
    margin-bottom: 10px;  
}  
  
.footer {  
    font-size: 10px;  
}
```

The `home-page.xml` page then used the classes to make the page header label large (32px), the footer label small (10px), center both labels, and provide a 10px margin on the top and bottom (listing 10.2).

Listing 10.2 An excerpt from home-page.xml at the end of Chapter 9

```
<StackLayout>  
    <Label class="header" text="{{title}}" />  
    <Image src "~/images/home.png"></Image>  
    <Label class="footer" text="{{footer}}" />  
    <StackLayout orientation="horizontal" horizontalAlignment="center" >  
        <Button style="margin-right: 20px;" text="About" />  
        <Button style="margin-left: 20px;" text="Continue"  
            tap="onContinueTap" />  
    </StackLayout>  
</StackLayout>
```

Before we go any further, delete the custom styles from the `home-page.css` file. We'll be using built-in styles from the theme package from here on.

WARNING If you don't delete the existing styles, you won't run into any errors, but your app may look different from ours.

10.2.1 Affecting text size

Let's start replacing the CSS by addressing the header font size. Labels used as headings can be replaced with *h1*, *h2*, ..., and *h6* CSS classes, like *<h1>*, *<h2>*, ..., and *<h6>* tags in HTML.

TIP Use *h1*, *h2*, *h3*, *h4*, *h5*, and *h6* CSS classes to style labels intended to be page headings.

The header label is intended to be the primary heading on the page, so let's change its class to an *h1*:
<Label class="h1" ... />.

Another common need is to create a picture caption or footnote, typically with a font size that is slightly smaller and less pronounced than normal page text.

TIP Use the *footnote* CSS class to style labels used as an image caption or footnote.

Our existing footer class is a great candidate for replacement with the built-in *footnote* class, so change the label to: *<Label class="footnote" ... />*.

10.2.2 Aligning text

Even though it doesn't take much effort to create a CSS rule to center-align labels by using *text-align: center*, we can use the built-in CSS class *text-center* to accomplish the same thing. So, let's update the two labels by adding the *text-center* class:

```
<Label class="h1 text-center" text="{{ title }}" />
<Label class="footnote text-center" text="{{ footer }}" />
```

TIP Use the built-in CSS classes *text-center*, *text-left*, and *text-right* to align labels in your app.

Last, we can clean up our use of margin and padding CSS rules using the built-in class. The built-in padding and margin classes use a convention-based approach of: *{margin/padding}-{{top/bottom/left/right}}-{amount}*, where the various keywords (like margin, top, and left) are abbreviated by using the first letter only. So, a CSS class of *m-t-25* adds a 25px margin top the top of an element. Likewise, *p-b-5* adds a 5px padding to the bottom of an element.

Let's replace our existing use of the top, bottom, left, and right margin rules with the new convention-based classes we just learned about. The first candidates are the About and Continue buttons with inline style rules:

```
<Button class="m-r-20" text="About" />
<Button class="m-l-20" text="Continue" tap="onContinueTap" />
```

By using the `m-r-20` and `m-l-20` classes, the a 20px margin was added to the right of the About button, and a 20px margin was added to the left of the Continue button.

To replace the style rules giving a 10px margin to the top and bottom of labels, we could use the built-in classes `m-t-10` and `m-b-10`, but there's a shortcut.

TIP When you want to apply the same margin or padding to an element on the same axis (top/bottom is considered the y-axis, left/right is the x-axis), you can use a single class. Substitute x or y for the directional component of the class. For example, `m-t-10` and `m-b-10` can be combined into `m-y-10`.

Using this new convention, we can quickly apply a margin of 10px to the top and bottom of the labels by adding the class `m-y-10`:

```
<Label class="h1 text-center m-y-10" text="{{ title }}" />
<Label class="footnote text-center m-y-10" text="{{ footer }}" />
```

10.3 Styling buttons

The last change we'll make to the home page is to our About and Continue buttons. One of the great things about NativeScript is that placing a button on the UI creates a native Android and iOS button - which includes the default styling of the native button. So, you may ask, why would I want to style my button differently?

In many cases, you may not want to. The native look and feel is exactly what you want. After all, Android apps should look and feel like Android apps, and iOS apps should look and feel like iOS apps. At the same time, we've talked to a lot of developers and we've heard the following three things:

4. Providing a unified look and feel to all their apps is important.
5. The default iOS button doesn't look like a button (it looks like a link) and often confuses developers new to iOS development.
6. App users frequently complain about button sizes being too little, and when they tap the button, they often miss the *hit box* of the button, causing them to tap the button multiple times to get it to work.

DEFINITION A button or link's hit box is the screen area around the button that detects whether a user has tapped the button. Smaller buttons (which inherently have smaller hit boxes) are more difficult to tap. Creating larger buttons is one way to increase the likelihood that users will tap the button on their first try.

By using the button classes of the built-in themes, you can address the three points above. Let's start by adding the base CSS class, `btn`, to our buttons:

```
<Button class="btn m-r-20" text="About" />
<Button class="btn m-l-20" text="Continue" ... />
```

This base class applies the default styles of a button, including size and spacing between with elements.

TIP Although it's not requirement, we recommend you start with this class when styling buttons.

The next choice you have is whether the button is a solid color or transparent. The *btn-primary* class styles a button with the theme's primary color, and *btn-outline* applies a transparent background with a thin outlined border. We like the look of colored buttons, so let's add the *btn-primary* class to our buttons:

```
<Button class="btn btn-primary m-r-20" text="About" />
<Button class="btn btn-primary m-l-20" text="Continue" ... />
```

WARNING Don't apply both the *btn-primary* and the *btn-outline* classes to the same button. Choose one.

The next button styling option you can choose from is whether to round the corners of your button. If you're going for a smoother design, you might like rounded edges, but if your app's design has more hard edges and angles, square corners might be better.

The default button style is square corners. To add rounded corners, use the *btn-rounded-sm* or *btn-rounded-lg* classes, which add small (sm) or large (lg) rounded corners to a button. We like the look of small rounded corners, so we've added these to our buttons:

```
<Button class="btn ... btn-rounded-sm" text="About" />
<Button class="btn ... btn-rounded-sm" text="Continue" ... />
```

Last, you can add a special effect to buttons that makes them appear highlighted to when they're tapped by adding the *btn-active* class. We like the subtle effect, so we've added it to the buttons:

```
<Button class="btn ... btn-active" text="About" />
<Button class="btn ... btn-active" text="Continue" ... />
```

10.3.1 Cleaning up

We've made a lot of changes to the home page, and learned how to style text and buttons. Before we check out the changes, delete the *home-page.css* file; we don't need it because we've replaced its entire functionality with built-in classes!

If you've been following along, the *home-page.xml* file should look like listing 10.3 and your app should look like figure 10.4. Great work!

Listing 10.3 An excerpt from *home-page.xml* after refactoring with built-in theme classes

```
<StackLayout>
    <Label class="h1 text-center m-y-10" text="{{ title }}" />
    <Image src="~/images/home.png"></Image>
    <Label class="footnote text-center m-y-10" text="{{ footer }}" />
    <StackLayout orientation="horizontal" horizontalAlignment="center" >
        <Button class="btn btn-primary btn-rounded-sm btn-active m-r-20"
            text="About" />
```

```
<Button class="btn btn-primary btn-rounded-sm btn-active m-l-20"  
       text="Continue" tap="onContinueTap" />  
</StackLayout>  
</StackLayout>
```

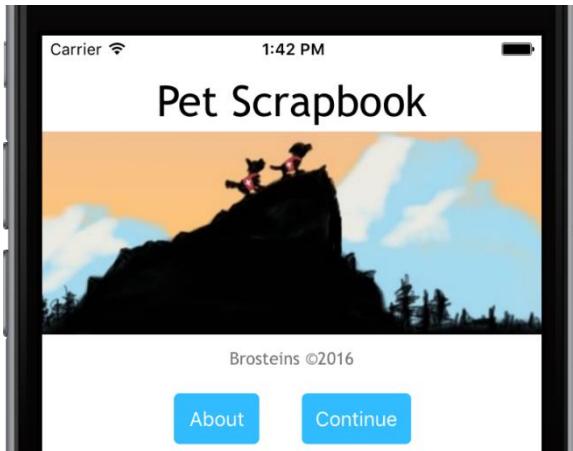


Figure 10.4 The refactored home page of the Pet Scrapbook, using built-in theme classes.

10.4 Styling list views

Now that we've refactored the home page of the Pet Scrapbook to use the built-in CSS classes, let's move on to the scrapbook page. As you'll recall, this page is a list of scrapbook pages (or entries). Listings 10.4, 10.5, and figure 10.5 show the relevant code from the scrapbook page after chapter 9.

Listing 10.4 An excerpt from `scrapbook-page.xml` after chapter 9

```
<ListView items="{{ pages }}" itemTap="onItemTap">  
  <ListView.itemTemplate>  
    <StackLayout orientation="horizontal">  
      <Image src="{{ image }}"/>  
      <Label text="{{ title, title + ' Scrapbook Page' }}"/>  
    </StackLayout>  
  </ListView.itemTemplate>  
</ListView>
```

Listing 10.5 Styles form `scrapbook-page.css` after chapter 9

```
label {  
  text-align: center;  
  margin-top: 10px;  
  margin-bottom: 10px;  
}  
image {  
  height: 50px;  
  width: 50px;
```

```
}
```

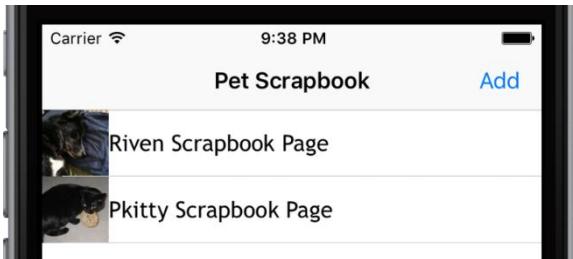


Figure 10.5 The scrapbook page at the end of chapter 9.

We didn't entirely ignore style for this page, but let's agree we can do a little better. Just like text and buttons have built-in class rules, list views can be styled by using themes. List views are styled with the *list-group* and *list-group-item* classes. The *list-group* class is applied to the ListView element, and the *list-group-item* class is applied to the top-level layout container within the list view item template. Listing 10.6 shows how to apply these classes.

Listing 10.6 Adding list-group and list-group-item to scrapbook-page.xml

```
<ListView items="{{ pages }}" itemTap="onItemTap" class="list-group"> #A
  <ListView.itemTemplate>
    <StackLayout orientation="horizontal" class="list-group-item"> #B
      <Image src="{{ image }}"/>
      <Label text="{{ title, title + ' Scrapbook Page' }}"/>
    </StackLayout>
  </ListView.itemTemplate>
</ListView>
```

#A The *list-group* class is applied to the ListView element
#B The *list-group-item* class is applied to the item template's layout container

Before we show you the results, let's do a few more things, starting with the item template's label. The list view theme has two classes for text: *list-group-item-heading* and *list-group-item-text*. Together, they're intended to be used for heading and normal text within a list view. We don't really have a heading, so we'll stick with the normal text class applied to the label:

```
<Label text="{{ title, title + ' Scrapbook Page' }}"
  class="list-group-item-text" />
```

The last change we'll make is with our image. It's customary in mobile apps to have an image or icon next to text within a list view, and the list view theme doesn't disappoint.

TIP Apply the *thumb* class to an image within a list view item template to turn the image into a thumbnail.

Let's update our image to include the *thumb* class:

```
<Image src="{{ image }}" class="thumb" />
```

And we're finished! Figure 10.6 shows the results of adding a few theme styles.

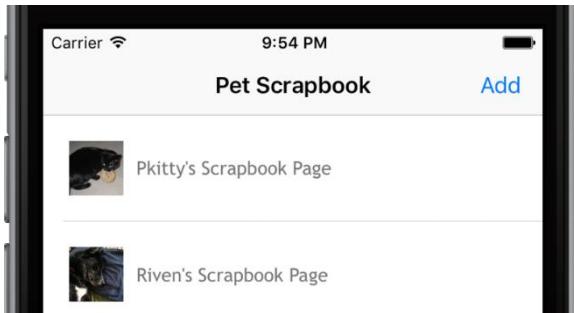


Figure 10.6 A refactored scrapbook-page.xml page incorporating list view theme styles.

10.5 Working with images

The Pet Scrapbook list view is good enough, but we can't resist adding a one final change to the thumbnail image. The built-in theme has two nifty classes: `img-rounded` and `img-circle` that allow you to add rounded corners to an image and even make the image circular. We're fans of the circular image look, so let's add the `img-circle` class to the thumbnail image:

```
<Image src="{{ image }}" class="thumb img-circle" />
```

WARNING Behind the scenes, the `img-rounded` and `img-circle` classes apply the `border-radius` CSS property to an image. `Border-radius` (and by extension `img-rounded` and `img-circle`) requires that the image have an explicit height and width set. Our example works because the `thumb` class explicitly set the width and height for us, but you should keep this in mind because it's easy to forget and stare at your CSS for hours wondering why it doesn't work. Hopefully, we can save you some time (and sanity)!

After adding the `img-circle` class, let's look (figure 10.7).

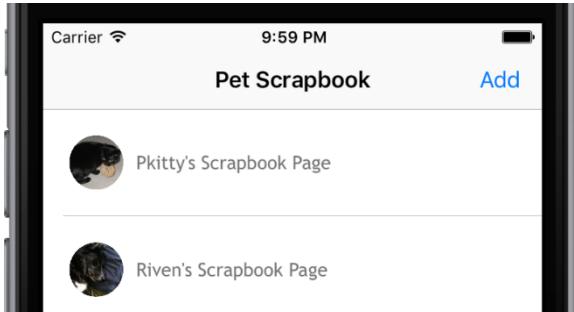


Figure 10.7 The scrapbook-page list view with a circular thumbnail image.

The scrapbook list view looks great! It's time to move on to the scrapbook update page, where you'll learn how to style the data entry form with a few more classes within the theme package.

NOTE Don't forget to delete the scrapbook-page.css file. Now that we've used the built-in themes, there's no need for the custom CSS styles.

10.6 Styling data-entry forms

So far, the theme styles we've applied to the Pet Scrapbook were quick changes that didn't require us to modify the UI structure. All we had to do was apply classes to an existing element. Well, we're about to change it up by adding several structural XML elements to help us refactor and style the scrapbook's update page. The modifications aren't extensive or really that invasive, but we'll be making some changes. Stick with us because it'll be worth it.

The Pet Scrapbook's update page is intended to be a simple data-entry form, but the product of chapter 9 is a far stretch from simple (figure 10.8).



Figure 10.8 The scrapbook update page, as built in chapter 9.

We can do better. Listing 10.7 is our starting point – the *scrapbookUpdate-page.xml* at the end of chapter 9.

Listing 10.7 The scrapbookUpdate-page.xml file at the end of chapter 9

```
<Page loaded="onLoaded">
  <StackLayout>
    <Label text="{{ title, title + ' Scrapbook Page' }}" />

    <TextField class="header" text="{{ title }}" hint="Enter title..."/> #A

    <Label text="{{ 'Age: ' + calcAge(year, month, day) +
      ' years old'}}" /> #B
    <DatePicker year="{{ year }}" month="{{ month }}" day="{{ day }}" /> #B

    <ListPicker items="{{ genders }}" selectedIndex="{{ gender }}" /> #C

    <Image src="{{ image }}" stretch="none" /> #D
    <StackLayout orientation="horizontal"> #D
      <Label text="{{ (lat, long), 'Picture taken at ' + #D
        lat + ', ' + long }}" /> #D
    </StackLayout> #D

    <Button tap="onAddImageTap" text="Add Image" />
    <Button tap="onDoneTap" text="Done"/>
  </StackLayout>
</Page>
#A Page's title (or name) data entry
#B Birth date selection
#C Gender selection
#D Picture selection
```

Like most data entry pages, the update page is separated into discrete areas for each data point we're collecting: the title, birth date, gender, and picture. But, there's a problem – everything blends together in the UI. It's not very user-friendly. Let's fix that by using the data form theme classes.

As we mentioned earlier, the data form theme classes impose an XML structure on your UI, to ensure uniformity and consistency. In summary, a data form consists of a primary stack layout and a series of child stack layouts. The primary layout is assigned the class *form*, and each series of child layouts is assigned the class *input-field*. Let's see this in action by wrapping each of our data entry fields with this structure (listing 10.8).

Listing 10.8 Refactored scrapbookUpdate-page.xml file using the form theme classes

```
<Page loaded="onLoaded">
  <StackLayout>
    <Label text="{{ title, title + ' Scrapbook Page' }}" />

    <StackLayout class="form"> #A
      <StackLayout class="input-field"> #B
        <TextField class="header" text="{{ title }}" #
          hint="Enter title..."/> #B
      </StackLayout> #B
```

```
<StackLayout class="input-field"> #B
    <Label text="{{ 'Age: ' + calcAge(year, month, day) +
        ' years old'}}" /> #B
    <DatePicker year="{{ year }}" month="{{ month }}"
        day="{{ day }}" /> #B
</StackLayout>
...
<Button tap="onAddImageTap" text="Add Image" />
<Button tap="onDoneTap" text="Done"/>
</StackLayout>
</StackLayout>
</Page>
```

#A Primary stack layout is added with the form class

#B Each data field label and control is wrapped in the input-field class

#C Remaining fields not shown, but would be wrapped in a stack layout with the input-field class

We made several structural changes to the update page, but they are straightforward. We added a primary stack layout with the form class. This stack layout encompasses the data entry form. Within the primary stack layout, we wrapped each data entry field with another stack layout. The input-field class is then applied to the stack layout.

This doesn't change the look of the page radically, as you can see in figure 10.9, additional space is added between and around each element with an input-field class applied.

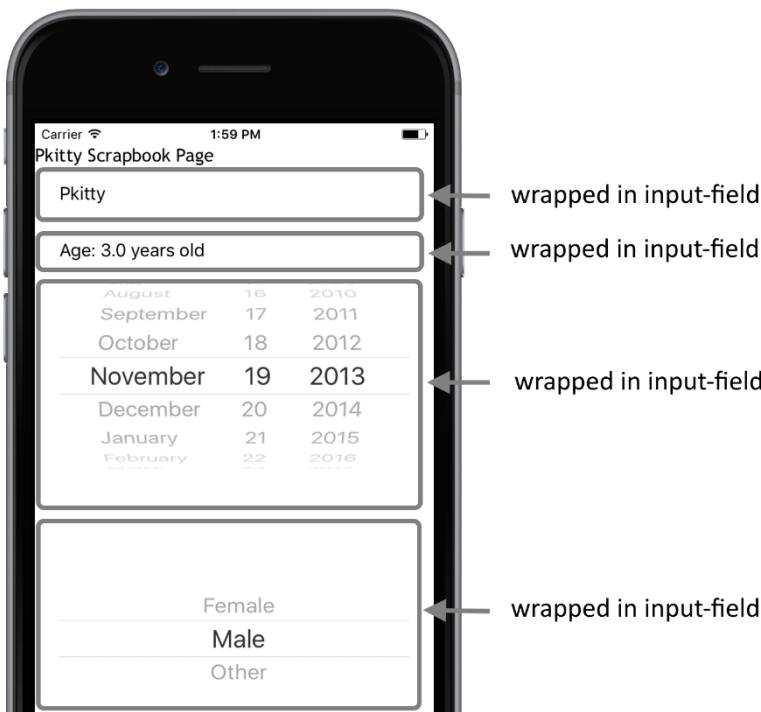


Figure 10.9 The scrapbook update page with data-entry fields wrapped in a stack layout with the input-field class applied.

10.6.1 Structuring data entry fields

We're almost there, but there's something missing. Standard data entry fields are composed of two elements: a label to describe the data entry field and a data entry control (like a text field, date picker, and so on).

TIP When creating a data entry page, combine a label and the data entry element together to create a consistent UI.

We're not following this pattern – look at the title, birth date, and gender fields. How is a user supposed to know the date picker is the birth date? Some users may not have trouble connecting the age label with the date picker below, but uniformity in the UI creates a consistent user experience.

Let's fix this by adding in consistent labels for the title, birth date, and gender, and image fields. As we're adding labels to the input field stack layouts, we'll apply two classes: *label* and *input*.

DEFINITION The label and input classes are used to style input field labels and text fields. They are part of the theme styles and help to create a more consistent UI experience.

Starting with the title field, we'll add a label with the *label* class. We'll also change the name of the field to *name* because it's a little more descriptive:

```
<Label class="label" text="Name:" />
```

Next up is the birth date label. The existing label displays the age, but it could be confusing. Let's call it what it is – the birth date, but also display the calculated age (listing 10.9).

Listing 10.9 Updated birth date label in the scrapbookUpdate-page.xml file

```
<Label class="label"
      text="{{ (year, month, day), 'Birth date: ' + #A
      (year === null ? #B
       ' ' :
       (' + calcAge(year, month, day) + ' years old)' ) }}" #C
/>
#A Always display the birth date field label
#B If the data bound year field is null, display nothing
#C Conditionally display the calculated age when a date is selected
```

We're using a slightly more advanced data-binding expression for the label's text value, so let's break it down step-by-step. First, we always want to display *Birth date:*, so this is part of the data-binding expression. Second, we want to conditionally display the age, which is calculated with the *calcAge()* function. To conditionally display the age, we use the *ternary operator*.

DEFINITION The ternary operator is a concise way of executing an if-then-else statement by using a syntax of {conditional-statement} ? {evaluate-if-true} : {evaluate-if-

`false}.` For example, an if-then-else statement of `if x, then y, else z` is expressed as `x ? y : z` using the ternary operator.

TIP Avoid displaying undefined or null data values in your UI – it's distracting to users. Use the ternary operator in data-binding expressions to control the conditional display of data-bound fields.

The last two fields are like the name field. We'll add a label to into their input field stack layouts and apply the `label` class.

```
<Label text="Gender:" class="label" />
<Label text="Image:" class="label" />
```

While we're working on the image data entry field, let's re-use the footnote class you learned about earlier and change the longitude and latitude label into a footnote using the ternary operator to only display the location if the longitude and latitude aren't undefined (listing 10.10).

Listing 10.10 Refactored image location label using ternary operator in the scrapbookUpdate-page.xml file

```
<Label class="footnote"
text="{{ (lat, long),
(lat === undefined || long === undefined) ?
'':
'Picture taken at ' + lat + ', ' + long }}"
/>
```

If you've been following along, your update page should look like listing 10.11 and figure 10.10.

NOTE Before running your app, don't forget to remove the custom label style from the scrapbookUpdate-page.css file. If you don't, your app won't look like ours.

Listing 10.11 Refactored scrapbookUpdate-page page using form theme classes

```
<Page loaded="onLoaded">
<StackLayout>
<Label text="{{ title, title + ' Scrapbook Page' }}" />

<StackLayout class="form">
<StackLayout class="input-field">
<Label text="Name:" class="label" />
<TextField class="header" text="{{ title }}"
hint="Enter title..."/>
</StackLayout>

<StackLayout class="input-field">
<Label class="label"
text="{{ 'Birth date: ' + (year === null ?
'': '(' + calcAge(year, month, day) + ' years old') ) }}"/>
<DatePicker year="{{ year }}" month="{{ month }}"
day="{{ day }}"/>
</StackLayout>

<StackLayout class="input-field">
```

```
<Label text="Gender:" class="label" />
<ListPicker items="{{ genders }}" selectedIndex="{{ gender }}" />
</StackLayout>

<StackLayout class="input-field">
<Label text="Image:" class="label" />
<Image src="{{ image }}" stretch="none" />
<Label text="{{ (lat, long),
(lat === undefined || long === undefined) ?
'':
'Picture taken at ' + lat + ', ' + long }}" />
</StackLayout>

</StackLayout>

<Button tap="onAddImageTap" text="Add Image" />
<Button tap="onDoneTap" text="Done"/>
</StackLayout>
</Page>
```

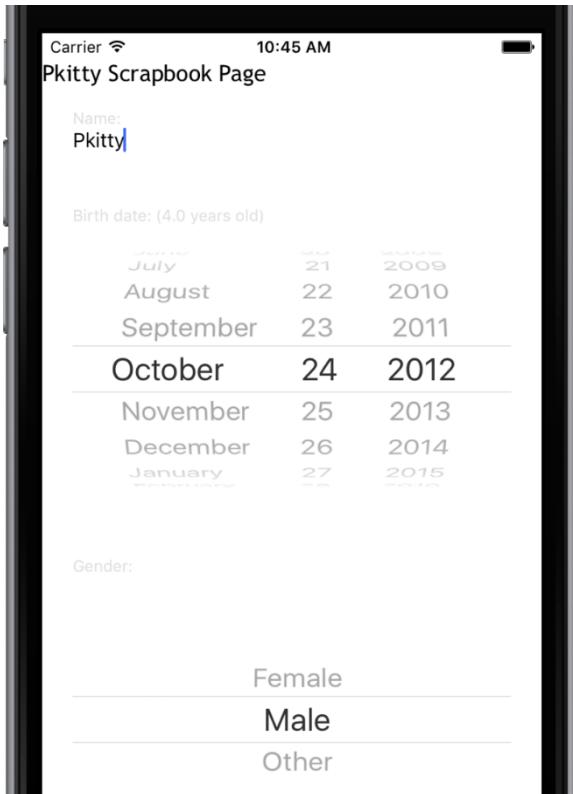


Figure 10.10 The update page refactored to use form theme classes and ternary operator data-binding expressions to conditionally display text.

We've covered a lot of ground with themes, and you've learned how to integrate the built-in theme classes into your app to save time and make more consistent UIs. There's still more to learn about themes, but it's awareness of additional classes for coloring text and other UI elements. We're not going to cover these specifically in this chapter, but we'll opportunistically introduce you to them throughout the rest of the book. If you're interested in learning more right now, the NativeScript documentation has a complete listing of every class included in the theme package. Check it out at <https://docs.nativescript.org/ui/theme>.

As we move away from themes, the update page still doesn't seem complete. In fact, it doesn't appear functional and looks cluttered. The birth date picker and gender list picker take up a lot of room, which causes users to scroll down the page. We can do better. In the next chapter, you'll learn how to clean up the UI and make it more functional.

10.7 Summary

In this chapter, you learned the following:

- You can style your app with a package of pre-built CSS styles by installing the nativescript-theme-core npm package.
- CSS margin and padding can be applied to elements with the a short-hand class name syntax of {m/p}-{t/b/l/r}-{#}, where m/p stands for margin and padding, t/b/l/r stands for top, bottom, left, and right, and # is a number of pixels.
- Using the theme CSS classes *form*, *input-field*, *label*, and *input*, you can create consistent data entry forms.
- The ternary operator ({if-conditional} ? {return-if-true} : {return-if-false}) is a short-hand logical expression that can replace an if-then-else statement.

10.8 Exercises

In this chapter, you learned how to use themes to style the UI of your app quickly and easily. Use what you've learned from this chapter to do the following:

- Change the color scheme of your app from the light color scheme to the dark color scheme.
- On the detail page, use the ternary operator to show the text "Unknown Pet's Scrapbook Page" when the title field is null or undefined, and "{title}'s Scrapbook Page" when a value is entered for the title. For example, if "Pkitty" is entered into the title field, the page title should read, "Pkitty's Scrapbook Page".
- On the detail page, make the title label standout by centering it and making it a heading (h1). Also add a 10-pixel margin to the y-axis.

10.9 Solutions

- To change the color scheme of the Pet Scrapbook from the light color scheme to the dark color scheme, replace the import statement at the top of the app.css file with a reference to the dark color scheme: `@import 'nativescript-theme-core/css/core.dark.css';`.
- To update the detail page's title label, replace it with: `<Label text="{{ title, (title ===`

```
null || title === undefined ? 'Unknown Pet' : title + '\\\'s') + ' Scrapbook  
Page'" />
```

- To make the title label stand out, add the following class to the label: `class="h1 text-center m-y-10"`.

11

Refining user experience

This chapter covers

- Improving user experience by moving date and list pickers to modal dialogs
- Targeting multiple screen resolutions

In the last chapter, you learned how NativeScript themes can make your apps look more professional with a minimal amount of effort. Using the Pet Scrapbook as an example, we styled the home, scrapbook list, and update pages with the built-in light color scheme. Even though the app has started to look more professional, the UI and functionality of the update page is clunky.

In this chapter, you'll learn how to create a cleaner and more concise UI by using modal dialogs. We'll also tackle the problem of multiple screen sizes, and when we're finished, we'll have a great looking app for both phones and tablets. Let's go!

11.1 Building professional UIs with modals

In chapter 10, you learned how to replace custom CSS styles with the built-in theme classes. We left off with the update page partially complete (figure 11.1).

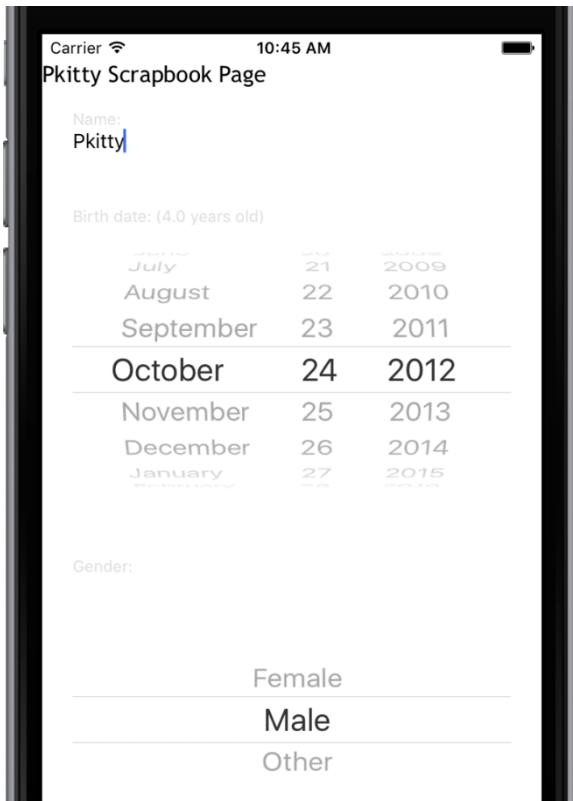


Figure 11.1 The update page refactored to use theme classes.

We think the update page is incomplete because the birth date picker and gender list picker take up too much room, creating a cluttered appearance. In mobile apps, appearance matters, so we're going to build a more professional UI by using *modal dialogs*.

DEFINITION Modal dialogs are a UI design concept where a user's interaction on a page prompts the UI to temporarily display a second page on top of the first page. When the user completes their interaction with the second page, it disappears and the UI is redirected to the first page. This UI interaction pattern (and the second page that is displayed) is referred to as a modal dialog because the second page often contains a dialog box or other UI element that is too big to fit on the first page.

Modal dialogs (a.k.a. modal pages or simply modals) are a great way to hide complex UI interactions on a page, and they display only when they're needed. For this reason, we'll be using modals to replace the birth date picker and gender list picker UI elements on the update page.

The goal of these changes is to create a less-complex UI that can fit on to the screen without the need for scrolling. To support the use of modals, the user's workflow and page navigation will change. Instead

of using the date picker and list picker directly on the update page, the user will tap a text field to enter the birth date and gender. When the text fields are tapped, a modal dialog will appear, containing the birth date picker and gender list picker. When they have selected the desired values, the modal dialog is closed and the selected values are displayed on the update page. Figure 11.2 shows a preview of this workflow for the birth date picker.

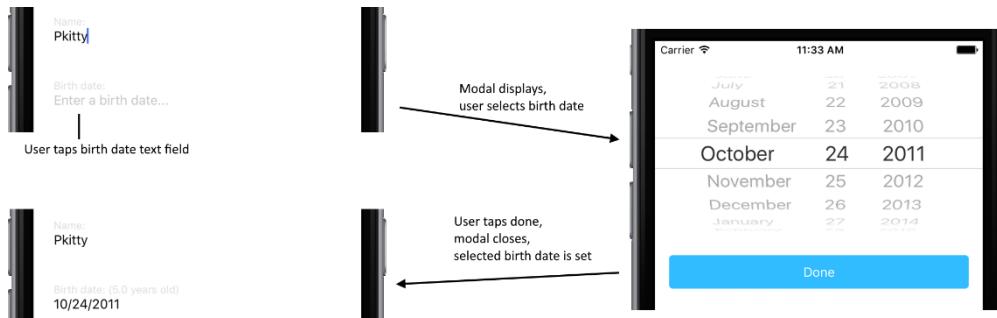


Figure 11.2 A user's interaction with the update page with modal dialogs used to select the birth date.

We'll reference this workflow as we refactor and build out new modals for the birth date picker and the gender list picker. Let's get started!

11.1.1 Moving date pickers to modal pages

Before we get knee-deep in code and UI refactoring, let's take a moment and outline our approach to this refactoring.

TIP When you're about to undertake a large refactor, it's a good idea to outline the steps you'll take. This will help collect your thoughts and document your plan. Long-term, your plan will help you if you need to step away for an extended period, as well as provide a coordination/collaboration point if you're working with others. Also, don't feel bad if your plan changes half-way through – just update your plan and continue forward.

Our refactoring plan has eight steps. Wow! Eight steps! Yeah, we know - it's a lot. If you're feeling overwhelmed, or if you're not sure you could have come up with these steps on your own, that's ok. We're going to go through each step, explaining it in detail.

1. Move the birth date picker element to a new page.
2. Replace the birth date picker with a text field, and prevent users from editing the text – our underlying page code will set the field value via data binding (step 7).
3. Replace the *month*, *day*, and *year* properties of the scrapbook observable with the *birthDate* property

4. Add a *tap* event and event handler to the text field, navigating to the birth date picker modal page when tapped. Pass in a close callback function the modal page will call when it is closed to pass back data to the update page.
5. On the birth date picker modal page, handle the *shownModally* event, saving a reference to the close callback function.
6. Data bind the birth date picker on the modal page.
7. Add a *tap* event and event handler to the done button, calling the modal's close callback when tapped. This will pass back the selected birth date to the update page.
8. Handle the close callback method on the update page, updating the birth date text field's value by updating its data bound field.

Now that we have our plan, let's get started by moving the date picker to a new page.

STEP 1: MOVING THE DATE PICKER TO A NEW PAGE

Start by creating two new files in the app/views folder: *selectDate-page.xml* and *selectDate-page.js*. But wait a minute! We're creating a page for a modal? Yep. Modals are simply pages, and you can create them just like you create any other page. Because they're pages, the file-naming conventions for the XML, JavaScript, and CSS files apply.

WARNING We said modals are just like pages, but there are technically some subtle differences between modals and pages. Although you define modals like you define pages, modals raise new events when loaded and navigating away from modals works differently. We'll cover these differences as we go, but for now, you can treat them just like you would a page.

Let's continue by adding a page element, stack layout, date picker, and button to the UI of the new page, as shown in listing 11.1.

Listing 11.1 The *selectDate-page.xml* file

```
<Page shownModally="onShownModally" loaded="onLoaded"> #A
  <StackLayout>
    <DatePicker date="{{ date }}" /> #B
    <Button class="btn btn-primary btn-rounded-sm btn-active"
      text="Done" tap="onDoneTap" />
  </StackLayout>
</Page>
#A We'll be handling the shownModally and loaded events
#B Instead of binding to the year, month, and day properties, we can bind directly to the date property
You'll notice a few new things we've introduced on this page. The first is the shownModally event.
```

DEFINITION The *shownModally* event is raised when a page is displayed as a modal dialog. It's like a page's *onLoaded* event, but raised as soon as the modal is shown on the mobile device.

In addition to the *shownModally* event, we've changed the data-bound property on the date picker. On the update page, we used the *year*, *month*, and *day* properties:

```
<DatePicker year="{{ year }}"
    month="{{ month }}" day="{{ day }}" />
```

Data binding to these three properties isn't necessary, and the same thing can be accomplished by binding directly to the *date* property of the date picker:

```
<DatePicker date="{{ date }}" />
```

That's all there is to the UI of the date picker modal page. We'll come back to the JavaScript to wire up our events and data binding in a later step.

STEP 2: REPLACE THE DATE PICKER WITH A TEXT FIELD

Now that we've create a modal page with a date picker, we can remove the date picker from the update page. Remove it, then replace it with text field. While you're there, update the label's databinding expression to use *birthday* instead of month, day, and year (listing 11.2).

Listing 11.2 The text field replacing the birth date data picker in scrapbookUpdate-page.xml

```
<Label text="{{ (birthDate), 'Birth date: ' +
(birthDate === null ?
'':
'(' + calcAge(birthDate) + ' years old)' ) }}"
class="label" />
<TextField class="input" editable="false"
text="{{ birthDate, birthDate | dateConverter(dateFormat) }}" #A
tap="onBirthDateTap" hint="Enter a birth date..." />
#A dateConverter is a type converter data-binding expression used to convert the birth date from a date to a string
```

On the surface, we're adding a typical text field, but there are a few new concepts introduced – the first being the *editable* property.

DEFINITION The *editable* property of a text field, when set to false, prevents users from tapping the text field and changing the value. This is similar to disabling a text field, but disabling changes the color of text to grayed out within the text field. When the *editable* property is set to false, users can't edit the content, and the content doesn't appear greyed-out.

We've also introduced a new type of data-binding expression: {{ birthDate, birthDate | dateConverter(dateFormat) }}. This type of data-binding expression contains something called a *converter*.

DEFINITION Converters solve a very specific problem during two-way data binding: conversion between two data types. More specifically, data within a data-bound observable may be stored as a complex object (for example., a date), but the date value may be bound to a text field, which displays a string. Converters provide a way to convert between these two objects.

Looking at our example, the *birthDate* property of our data-bound observable is a date, and the *text* property of the text field displays a string. On its own, NativeScript data binding doesn't know how to convert between two different objects. It'll guess, but often it's wrong. Think about it for a minute: there are dozens of ways to display a date: US format (MM/DD/YYYY or MM/DD/YY), ISO format (YYYY-MM-

DD), and so on. To solve this problem, you create a converter function that explicitly describes how to convert the values.

Internationalization (i18n) and NativeScript

You shouldn't assume your apps will be used in a single country and support a single language. In fact, your app may be used all over the world.

The process of developing your app so it can be easily adapted to support multiple languages and cultures is called internationalization (i18n).

Although we're not going to focus on i18n in this book, there is an npm plugin named nativescript-i18n that implements i18n. By using this plugin, you can build your app so it can support multiple languages and cultures. If your app requires support for i18n, or you want to build an app that is accessible to multiple languages and cultures, check out this plugin at <https://www.npmjs.com/package/nativescript-i18n>.

You can tell that the `{} birthDate, birthDate | dateConverter(dateFormat) {}` data-binding expression uses a converter because of the vertical bar | symbol. Functions following the vertical bar are converter functions.

Now that you know what a converter function is, you need to know how to define a converter function. They're defined like normal functions but need to be registered globally in the `app.js` file. Let's look at the function definition in our `app.js` file (listing 11.3).

Listing 11.3 Converter function for converting dates to MM/DD/YYYY format, defined in the `app.js` file

```
var application = require("application");
application.cssFile = "./app.css";

var dateConverter = function(value, format) { #A
    if (value === null || value === undefined || value === '') return ''; #A
    var parsedDate = new Date(value); #A
    #A
    var result = format;
    var day = parsedDate.getDate(); #A
    result = result.replace("DD", day); #A
    var month = parsedDate.getMonth() + 1; #A
    result = result.replace("MM", month); #A
    result = result.replace("YYYY", parsedDate.getFullYear()); #A
    return result; #A
};
var resources = application.getResources(); #B
resources.dateConverter = dateConverter; #C
resources.dateFormat = "MM/DD/YYYY"; #D

application.start({ moduleName: "views/home-page" });
```

#A Converter functions take in two parameters: the value to convert and the format (or options) to use during conversion

#B Application resources are a global object collection

#B Converter functions should be registered globally in the resources collection

#C Also register the format string used globally so we can share it throughout the app

Converter functions are defined like a normal function and take two parameters: the *value* to convert and a *format* (or *options*) string that can be used to specify how to format the value. The `dateConverter()` function expects a format string with the letters MM, DD, and YYYY inside. When run, it uses the format string to replace MM, DD, and YYYY with the month, day, and year.

To use the converter function in a data-binding expression, it needs to be registered globally in the application resources collection. While we're registering objects globally, we add an entry for the date format (MM/DD/YYYY) that we want to use within data-binding expressions.

TIP If you'll be using a date format more than once in your app, it's a good idea to register it globally.

If you ever need to change it, you have a single place to go to update it throughout your app.

After making the changes to the `app.js` file and the `scrapbookUpdate-page.xml` file, the update page should look like figure 11.3. If you're following along, go ahead and try to tap the birth date text field. You shouldn't be able to because it's not editable!

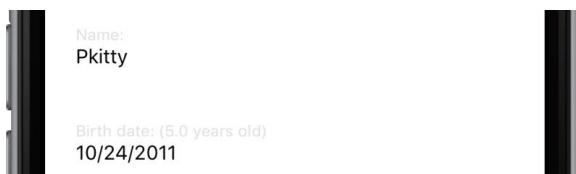


Figure 11.3 The birth date input field changed from a date picker to a text box.

STEP 3: REPLACE THE MONTH, DAY, AND YEAR PROPERTIES OF THE SCRAPBOOK OBSERVABLE WITH THE BIRTHDATE PROPERTY

After adding the text field to the update page, you'll need to update the scrapbook observable to use the `birthDate` property, instead of the `month`, `day`, and `year` properties. Update the `scrapbookPageModel()` and `onLoaded()` functions in the `scrapbook-page.js` file with the code from listing 11.4.

Listing 11.4 Updated scrapbookPageModel() and onLoaded() functions in the scrapbook-page.js file to support databinding to the birthDate property

```
function scrapbookPageModel(id) {
    var model = new observable.Observable();
    model.id = id;

    model.genders = ["Female", "Male", "Other"];
    model.calcAge = function(birthDate) {
        var now = Date.now();
        var diff = Math.abs(now - birthDate) / 1000 / 31536000;

        return diff.toFixed(1);
    }
}
```

```

};

return model;
}

exports.onLoaded = function(args) {
    var page = args.object;
    var scrapbook = new observable.fromObject(
        { pages: new observableArray.ObservableArray() });
    var pages = fileSystemService.fileSystemService.getPages();

    if (pages.length !== 0) {
        pages.forEach(function (item) {
            var model = new scrapbookPageModel();

            model.id = item.id;
            model.title = item.title;
            model.gender = item.gender;
            model.birthDate = item.birthDate;
            model.image = item.image;
            model.lat = item.lat;
            model.long = item.long;

            scrapbook.pages.push(model);
        });
    }
    else {
        scrapbook = new observable.fromObject({
            pages: new observableArray.ObservableArray()
        });
    }

    page.bindingContext = scrapbook;
};

```

The `savePage()` and `getPages()` functions of the file system service also needs updated. Use the code in listing 11.5.

Listing 11.5 Updated `savePage()` `fileSystemService.js` to support databinding to the `birthDate` property

```

fileSystemService.prototype.getPages = function () {
    var pages = [];

    if (this.file.readTextSync().length !== 0) {
        pages = JSON.parse(this.file.readTextSync());
    }

    pages.forEach(function (page) {
        page.birthDate = new Date(page.birthDate); // #A
        if (page.imageBase64 != null) {
            page.image = imageModule.fromBase64(page.imageBase64);
        }
    });
}

return pages;
}

```

```

fileSystemService.prototype.savePage = function (scrapbookPage) {
    var pages = this.getPages();

    var index = pages.findIndex(function (element) {
        return element.id === scrapbookPage.id;
    });

    if (index !== -1) {
        pages[index] = {
            id: scrapbookPage.id,
            title: scrapbookPage.title,
            gender: scrapbookPage.gender,
            birthDate: scrapbookPage.birthDate,
            imageBase64: scrapbookPage.image != null ?
                scrapbookPage.image.toBase64String("png") : null,
            lat: scrapbookPage.lat,
            long: scrapbookPage.long
        };
    }
    else {
        pages.push({
            id: scrapbookPage.id,
            title: scrapbookPage.title,
            gender: scrapbookPage.gender,
            birthDate: scrapbookPage.birthDate,
            imageBase64: scrapbookPage.image != null ?
                scrapbookPage.image.toBase64String("png") : null,
            lat: scrapbookPage.lat,
            long: scrapbookPage.long
        });
    }

    var json = JSON.stringify(pages);
    this.file.writeText(json);
};

#A create a new Date object when date is read from the file

```

STEP 4: NAVIGATING TO THE MODAL PAGE

After adding the text field to the update page, let's wire-up the tap event handler by adding the `onBirthDateTap()` function to the `scrapbookUpdate-page.js` file (listing 11.6).

NOTE We've also updated the `onLoaded()` function to store a reference to the page globally.

Listing 11.6 The updated `onLoaded()` function and new `onBirthDateTap()` function to navigate to the birth date selection modal page

```

var page;

exports.onLoaded = function(args) {
    page = args.object;
    var scrapbookPage = page.navigationContext.model;

    page.bindingContext = scrapbookPage;
};

```

```
exports.onBirthDateTap = function(args) {
    var modalPageModule = "views/selectDate-page";
    var context = { birthDate: page.bindingContext.birthDate };
    var fullscreen = true;

    page.showModal(
        modalPageModule,
        context,
        function closeCallback(birthDate) { #A
            page.bindingContext.set("birthDate", birthDate); #A
            #A
        },
        fullscreen
    );
};

#A function called by the modal page when it closes, and is used to pass data from the modal page back to this page
```

Navigating between the update page to the modal page is similar to navigating between regular non-modal pages, except you use the `showModal()` method of the page. The `showModal()` function takes four parameters:

- the destination page (`views/selectData-page`, which is the page we just created).
- the context (or data) you'd like to share with the modal page (we'll be passing in the birth date value stored in our page's binding context).
- a close callback function that will be called by the modal page when it is closed.
- a boolean value indicating whether the modal will be displayed full screen when opened.

In an earlier chapter, you learned about navigating between pages, so the `showModal()` should seem familiar, but the *close callback* function is something new.

DEFINITION The close callback function is a method called by the modal dialog when it's closed. In addition to closing the modal window, it is the mechanism for passing context (or data) back from the modal.

The close callback function may seem a bit confusing, but we're going to wait to fully explain it because it will make more sense when you see how it's used from within the modal page.

For now, here's what you should take-away: this function will be executed when the birth date selection modal page is closed, and the selected birth date will be passed back as an argument. When this happens, we'll update the birth date text field's `text` property by updating its `data-bound` property: `page.bindingContext.set("birthDate", birthDate)`.

With the show modal JavaScript code complete, tapping the birth date text field will open the birth date selection page, as shown in figure 11.4.

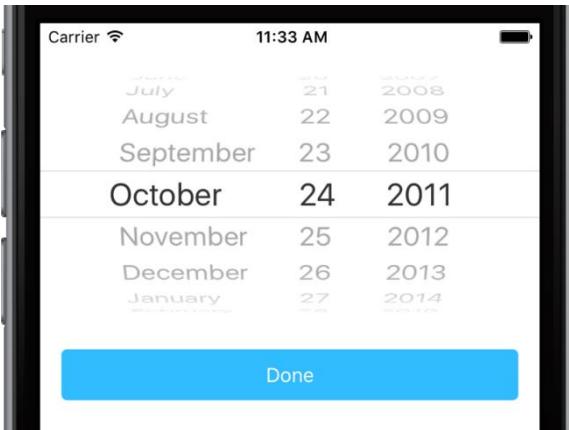


Figure 11.4 The birth date selection page shown modally after tapping on the birth date text field.

STEP 5: HANDLING THE MODAL'S SHOWNMODALLY EVENT AND LOADED EVENTS

If you're following along with our workflow, we've created a new modal page containing our date picker, replaced the data picker with a text field on the update page, and navigated to the modal page. In this step, we'll be initializing the modal page by handling the `shownModally` event.

We haven't written any JavaScript for the `selectDate-page.js` file yet, so let's look at the contents in listing 11.7. Add this to the `selectDate-page.js` file.

Listing 11.7 Responding to the shownModally event of the selectDate-page page

```
var closeCallback;

exports.onShownModally = function(args) { #A
    closeCallback = args.closeCallback; #A
}; #A

#A when the modal is shown, you have access to the close callback function, be sure to save a reference to it
```

When the modal page is shown, you have access to the close callback function through a property named `closeCallback`. We saved a reference to this method so we can call it when we're finished.

WARNING It's imperative that you store a reference to the close callback function when a modal is shown. Without a reference to the close callback, you'll be unable to close the modal and pass data to the page that opened the modal.

STEP 6: SET THE PAGE'S BINDING CONTEXT TO DATA BIND THE DATE PICKER

Now that we've saved a reference to the close callback, let's turn our attention to binding the date picker to an observable so we can easily get the selected value via code. Listing 11.8 adds to the `selectDate-page.js` file, establishing a binding context for the page.

Listing 11.8 Establishing a binding context when the selectDate-page page is loaded

```
var observableModule = require("data/observable");
```

```
var model;

exports.onLoaded = function(args) {
    var page = args.object;

    model = new observableModule.fromObject({
        date: new Date(Date.now())
    });
    page.bindingContext = model;
};
```

The loaded event handler is code you've seen before to create a new observable with date property. You'll recall from chapter 8 that setting the page's binding context enables you to use mustache syntax binding in XML.

Back in step 1, we created the UI for the birth date modal, which included a data-bound date picker: <DatePicker date="{{ date }}" />. By setting the binding context in listing 10.8 we've linked together the observable's *date* property with the date picker's *date* property. This means changes to the date picker's selected date value will be synchronized into the observable.

STEP 7: HANDLING THE DONE BUTTON'S TAP EVENT

Throughout this large refactoring, we've been promising to explain the close callback function and how it's used. If you recall, the close callback originates when it is passed as an argument to the `showModal()` function that shows the modal dialog (listing 11.9).

Listing 11.9 The origin of the close callback function as it is passed into the `showModal()` function

```
page.showModal(
    modalPageModule,
    context,
    function closeCallback(birthDate) {
        page.bindingContext.set("birthDate", birthDate);
    },
    fullscreen
);
```

We can follow the journey of the close callback through to the *shown modally* event's handler, where we saved a reference to it, as shown in listing 11.10:

Listing 11.10 Saving a reference to the close callback function when the birth date selection modal is shown

```
var closeCallback;

exports.onShownModally = function(args) {
    closeCallback = args.closeCallback;
};
```

Previously, we said it was critically important to save a reference to a modal's close callback function but didn't explain fully. Although modal dialogs are like pages, you don't navigate away from a modal as you do a page. Instead, modals are closed, returning control (and data) back to the previous page. We've been following the journey of the close callback because it's essential to closing a modal. In fact, the close callback is *how* to close a modal.

NOTE To close a modal dialog, call the close callback function, passing in any data that needs to be returned to the previous page.

Now that we've learned how the close callback works, let's flesh out the Done button's tap event handler by closing the modal dialog and passing back the selected date. Add the `onDoneTap()` function to the `selectDate-page.js` file:

```
exports.onDoneTap = function(args) { closeCallback(model.date); };
```

Closing a modal and passing data back is really that easy: one function does it all.

STEP 8: USING THE CLOSE CALLBACK TO UPDATE THE UI

We're almost there, and the last step is to update the UI by setting the `birthDate` property of the page's binding context to the birth date passed back by the modal dialog's close callback event.

There's actually nothing to do because we've already written the code to do this. When the close callback function was passed to the `showModal()` function, its body included a statement to update the `birthDate` property: `page.bindingContext.set("birthDate", birthDate);`.

And that's it - the refactoring is complete. We've simplified the UI by replaced the birth date picker with a text field and moved the date picker to a modal dialog. The result of the refactoring can be seen in figure 11.5.

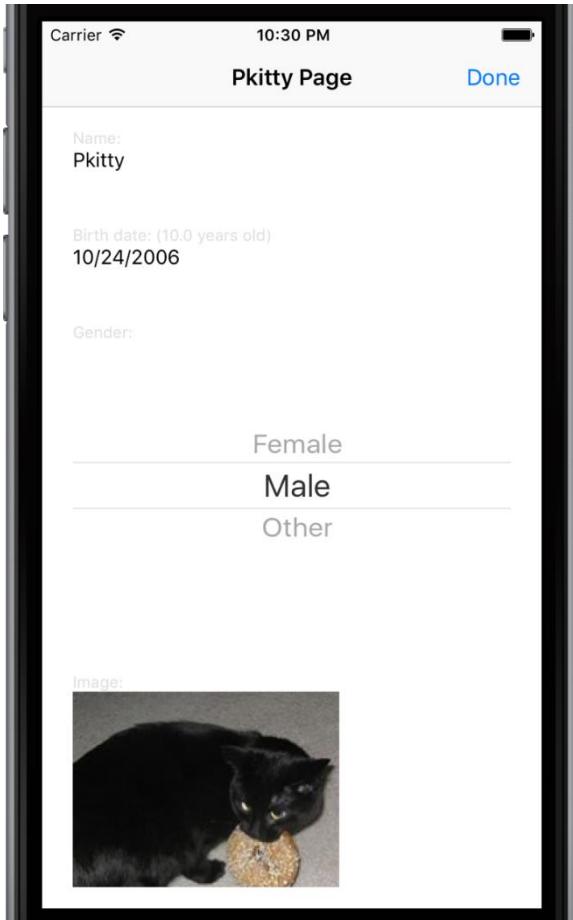


Figure 11.5 The scrapbookUpdate-page page with a refactored birth date input field.

HOMEWORK TIME

With the birth date input field updated, the next logical refactoring is the gender list picker. Refactoring the list picker to a modal dialog should be almost identical to the date picker, so we're going to challenge you to do the refactoring before the end-of-chapter exercise. We know you can do it, just start off with a plan. If you get stuck, don't worry: you can always peek at the code in our Github repository: <https://github.com/mikebranstein/TheNativeScriptBook/tree/master/Chapter11/PetScrapbook>.

When you've finished refactoring the list picker, your final page should look like figure 11.6.

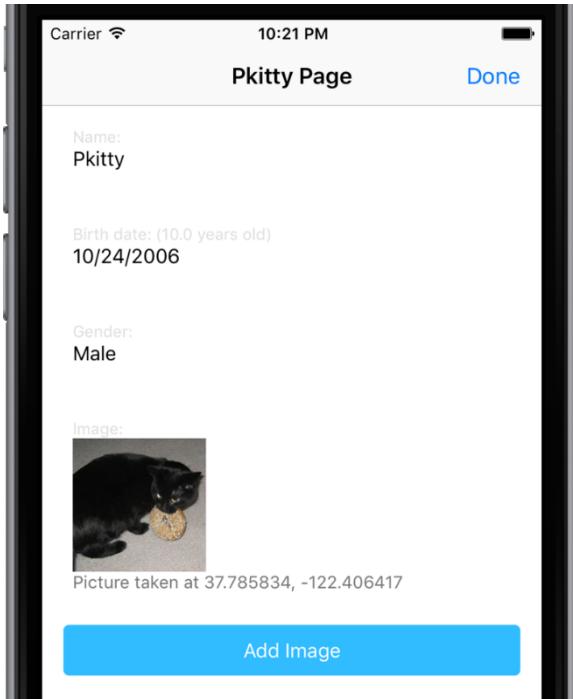


Figure 11.6 The scrapbookUpdate-page page with both date picker and list picker input fields refactored to modals.

11.2 Adding tablet support to an app

Throughout this chapter, you've learned how to create more consistent and professional-looking apps with themes and modal dialogs. With these tools, we transformed the Pet Scrapbook app into a more visually appealing app. The changes we made really helped make the app look more professional, but we failed to think about tablets. Imagine you had a tablet running the Pet Scrapbook. Now think about the page with a list of scrapbook pages. Would it make sense to have the list of scrapbook pages take up the entirety of a tablet's screen? Figure 11.7 shows how much space is wasted when the app is run on a tablet.

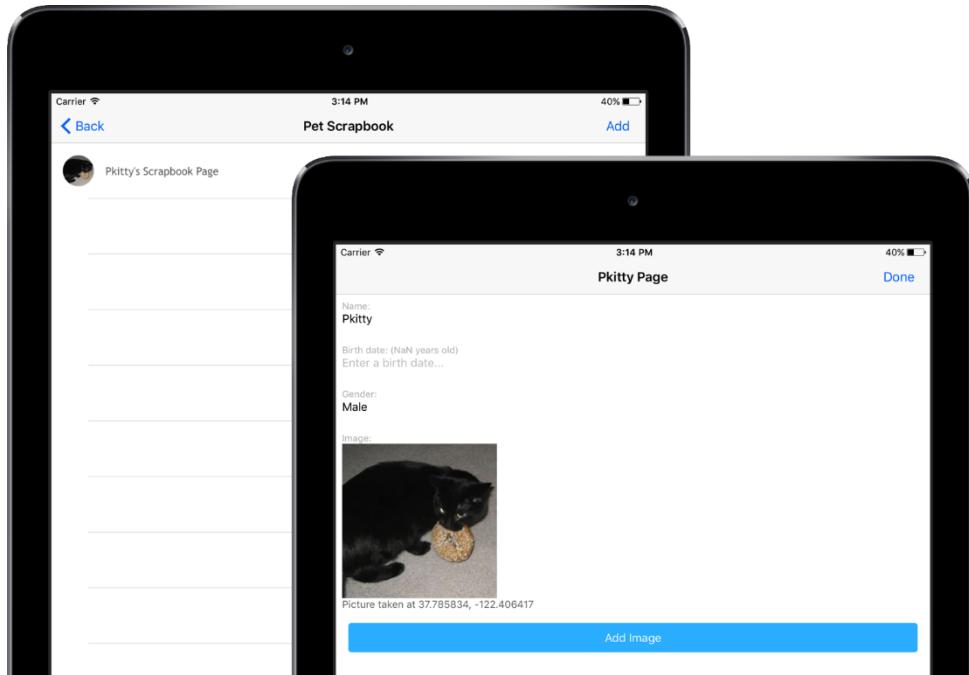


Figure 11.7 The Pet Scrapbook running on an iPad, note the large amount of unused space in the UI.

With a larger screen available, it feels wasteful to have the entire screen dedicated to the list of scrapbook pages. A better use of the available screen space is a split screen: the list of pages on the left, and selecting a page from the list displays the details on the right. Figure 11.8 shows the desired effect in action.

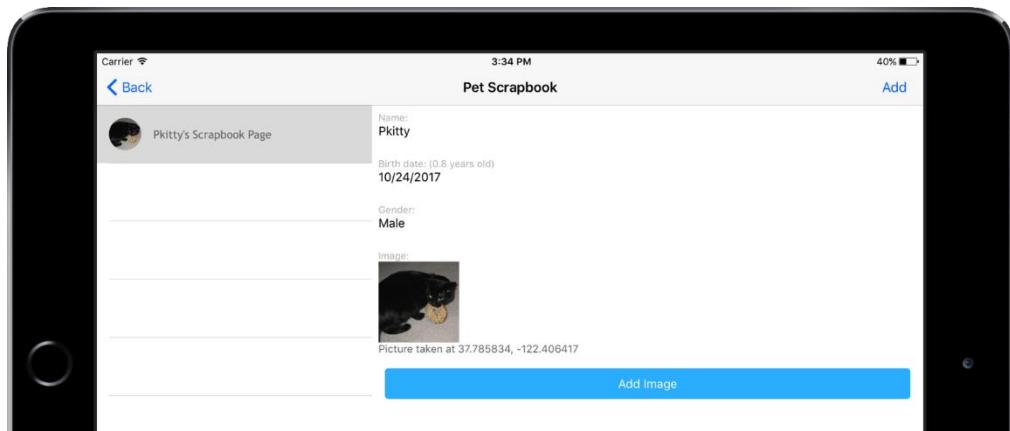


Fig 11.8 A landscape view of the Pet Scrapbook on an iPad, with the page list on the left and the detail view on the right.

In this section, we'll continue to make the Pet Scrapbook a more professional app by adding support for multiple screen resolutions.

11.2.1 Targeting multiple screen resolutions

The Pet Scrapbook started with three pages: the home page (`home-page.xml`), a list of all scrapbook pages (`scrapbook-page.xml`), and a scrapbook page where the details of a scrapbook could be updated (`scrapbookUpdate-page.xml`). When these pages are run on a mobile device, they will look the same regardless of the screen size.

Professional apps account for both phones and tablets, so we'd like to show a different list page when the app is run on a tablet. But, before we proceed with a tablet-specific version of the app, there's a few scenarios we should consider: using the app on a phone or tablet, and using the app in portrait or landscape. Table 11.1 breaks down these options.

Table 11.1 Scrapbook list and details page behaviors, based on device type and display orientation

Device type	Display orientation	Page layout plan
Phone	Portrait	Scrapbook list and details on separate pages (considered default behavior)
Phone	Landscape	Scrapbook list and details on separate pages
Tablet	Portrait	Scrapbook list and details on separate pages
Tablet	Landscape	Scrapbook list and details on the same page

In most scenarios (phone portrait and landscape, tablet portrait), the app's behavior should be exactly as we've seen already: the scrapbook list and details pages are separate pages. But, when running the app on a tablet in landscape mode, we want to use the extra horizontal screen real estate to show both the scrapbook list and the details of a selected scrapbook page.

You may recall how to target multiple screen resolutions and orientations from chapter 3, but it's been a while so let's review quickly. To target landscape and portrait orientations, you change the file name of your page and JavaScript files. You change the file name from `{page-name}.xml` to `{page-name}.land.xml` or `{page-name}.port.xml`. To target a tablet the file name changes from `{page-name}.xml` to `{page-name}.minWH600.xml`, where the additional `.minWH600` convention means the page should be loaded when a device's minimum width (`minW`) or height (`H`) is at least 600 device-independent pixels (dp).

NOTE We discussed device-independent pixels in chapter 3, which is a way of describing mobile device screen sizes in a standard way. Check out chapter 3 for more details, or read this blog post: <http://blog.fluidui.com/designing-for-mobile-101-pixels-points-and-resolutions/>.

NOTE A minimum width or height of 600 dpi isn't a number we decided to randomly use, but is a generally-accepted number of device-independent pixels for tablets.

Using these conventions, let's create a landscape-oriented, tablet-specific page for the scrapbook-page.xml by adding the XML and Javascript files scrapbook-page.land.minWH600.xml and scrapbook-page.land.minWH600.js to the views folder of the app. After making this addition, and adding a temporary label to the newly-created page, our app shows different pages when viewed on a phone and tablet (figure 11.9).

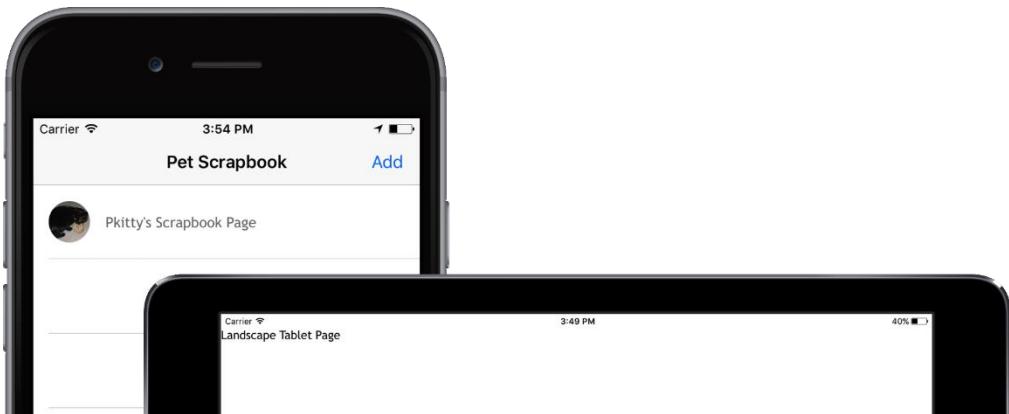


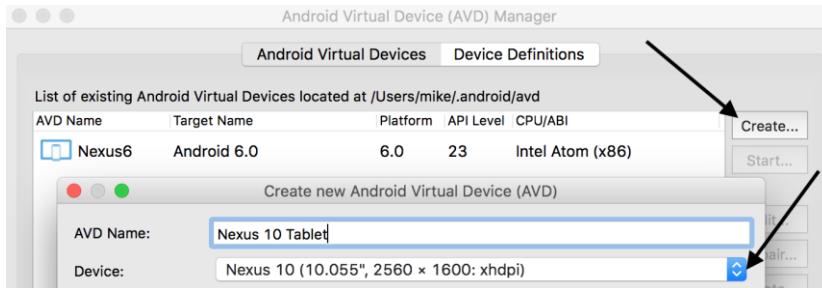
Figure 11.9 The scrapbook-page page on a phone and landscape tablet.

Testing tablet-specific apps on the Android emulator and iOS simulator

You may have wondered how you can test tablet-specific pages on the Android emulator and iOS simulator. Running and testing a NativeScript on a tablet emulator/simulator is no different than doing it on a phone-sized device. It may sound a bit over-simplified, but the only thing you need to do is run the tablet-sized device running instead of a phone-sized device.

Truthfully, running and testing on the tablet emulator/simulator isn't difficult, but it can be confusing the first time you do it. Don't worry, we'll walk you through it on the stock Android emulator, GenyMotion (the third-party Android emulator), and the iOS Simulator.

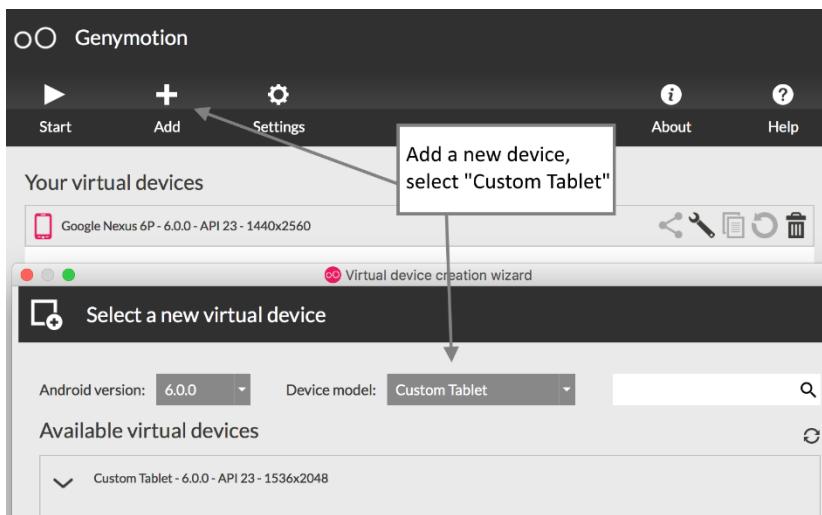
To setup a tablet in the stock Android emulator, open the Android Virtual Device (AVD) Manager and create a new device (note your AVD Manager may look slightly different).



To add a tablet, create a new virtual device, and select a tablet-sized device in the Android Virtual Device (AVD) Manager.

When creating the virtual device, the *Device* drop down will have a variety of devices to choose from. Select a tablet from the list. If you're not sure which ones are tablet, you could search online, or pick our go-to option: the Nexus 10. It doesn't *really* matter which tablet you choose, just make sure it's a tablet. After adding the tablet, start the emulator and run your app via the NativeScript CLI: `tns run android`.

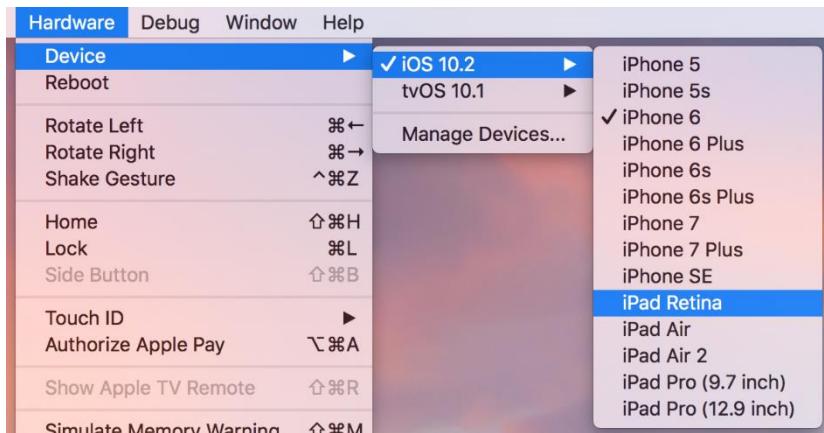
Configuring a tablet emulator on GenyMotion is like setting one up on the stock Android emulator. Open GenyMotion and press the large *Add* button, and select the *Custom tablet* option from the *Device model* drop down.



In GenyMotion, add a Custom Tablet device to install and run a tablet emulator.

Select one of the Custom Tablet virtual devices in the Available virtual devices list and complete the wizard to create your virtual device. After the tablet is added, start it up, and run your app via the CLI: `tns run android`.

The iOS simulator is a bit easier to configure a tablet. Start by running the Simulator app. When the simulator app launches, it will automatically start an iPhone simulator (or the last running device you used). To switch to a tablet-sized simulator, go to the Hardware menu, select Device, then iOS, and finally a tablet-sized device. Any of the iPads will do.



The iOS Simulator allows you to switch devices from the Hardware – Device – iOS menu.

After a new device is selected, the currently-running simulator will shut down and the device you selected will be loaded. To run your app on the device, use the CLI: `tns run ios`.

Now that we have a separate page for landscape-oriented tablets, let's get down to business. The tablet version will be like the existing app, and will be composed of three features:

9. A list of scrapbook pages. The left side of the screen will display a list of scrapbook pages, reusing the functionality we've defined already in the `scrapbook-page.xml` file.
10. A detailed view of a single scrapbook page. When a user selects a page from the list, the right side of the screen will display the detailed view of the selected page, reusing the UI and business logic from the `scrapbookUpdate-page.xml` file. From the detailed view the user will be able to save changes made to the scrapbook page.
11. Adding a new scrapbook page. Users will be able to add new pages to the list of scrapbook pages and update the page's content from the detailed view on the right.

CREATING THE BASE XML CODE STRUCTURE

Let's start by creating the structure for the new page, leaving a space for the list and details areas. Listing 11.11 outlines the base structure of the tablet-specific scrapbook page.

Listing 11.11 Initial structure of the scrapbook-page.land.minWH600.xml page

```
<Page loaded="onLoaded">
  <Page.actionBar>
    <ActionBar title="Pet Scrapbook" >
      <ActionItem tap="onAddTap" ios.position="right"
        text="Add" android.position="actionBar"/>
    </ActionBar>
  </Page.actionBar>
  <GridLayout rows="*" columns="*, 2*" >
    #A
    <GridLayout rows="*" columns="*" col="1" >
      <StackLayout> #B
        #B
      </StackLayout> #B
    </GridLayout>
  </GridLayout>
</Page>
#A The scrapbook list code goes here
#B The detailed view code goes here
```

The tablet-specific page starts with a similar action bar you saw earlier in this chapter, displaying the app's name and an action item that adds a new scrapbook page to the list of pages. The remainder of the page is organized with a single grid layout with a single row and two columns. Although you don't see the actual XML code for the list and detailed view in listing 11.11, we've called out the location of each. The scrapbook list will be placed in the first column, and consume one-third of the screen's width. The detailed view of a selected page will take up the right two-thirds of the screen and be wrapped inside of an additional grid layout and stack layout.

BUILDING OUT THE JAVASCRIPT CODE BASE

To wrap up the base structure of the page, let's lay down the JavaScript code to go along with the XML code we just added by starting with the page's loaded event (listing 11.12).

Listing 11.12 Handling the loaded event in the scrapbook-page.land.minWH600.js file

```
var observable = require("data/observable");
var observableArray = require("data/observable-array");
var fileSystemService = require("~/data/fileSystemService");

var page;

exports.onLoaded = function(args) {
  page = args.object;
  var scrapbook = new observable.fromObject({ // A
    pages: new observableArray.ObservableArray(), // A
    selectedPage: null // A
  }); // A

  var pages = fileSystemService.fileSystemService.getPages(); // B
```

```

if (pages.length !== 0) { // #B
    pages.forEach(function (item) {
        var model = scrapbookPageModel(); // #B

        model.id = item.id; // #B
        model.title = item.title; // #B
        model.gender = item.gender; // #B
        model.birthDate = item.birthDate; // #B
        model.image = item.image; // #B
        model.lat = item.lat; // #B
        model.long = item.long; // #B

        scrapbook.pages.push(model); // #B
    });
}

page.bindingContext = scrapbook; // #C
};

function scrapbookPageModel(id){ // #D
    var model = new observable.fromObject({
        id: id,
        title: null,
        birthDate: null,
        gender: null,
        image: null,
        lat: null,
        long: null
    });

    model.calcAge = function(birthDate) {
        var now = Date.now();
        var diff = Math.abs(now - birthDate) / 1000 / 31536000;

        return diff.toFixed(1);
    };

    return model;
}

#A The definition of the observable is new for the tablet-specific page, tracking the collection of pages and the selected page
#B This is almost identical, except the scrapbook pages are pushed to the scrapbook.pages observable array
#C Because the page contains both a list of all pages and the selected page, the binding context is set to the over-arching observable
#D No changes here

```

You'll recognize a bit of this code from earlier in this chapter, but it's been changed a little. When the tablet-specific page loads, we load the existing pages from the file system and establish a binding context for the page (this is assigned to an observable named *scrapbook*). The scrapbook observable will be used to track two things: an observable array of scrapbook pages (the *pages* property) and the selected scrapbook page (the *selectedPage* property).

NOTE Planning ahead, we'll be using the `pages` and `selectedPage` properties in data binding. When we've added the scrapbook list and detailed view UI elements to the page, we'll bind to these properties.

With the `loaded` event added, let's turn our attention to the action bar item that adds a new scrapbook page. Now that we have an observable array (`pages`), we'll add a function to handle the tap event and add a scrapbook page to the `pages` observable array (listing 11.13).

Listing 11.13 Adding a scrapbook page when the action bar action item is tapped

```
exports.onAddTap = function(args) {
    var page = args.object;
    var scrapbook = page.bindingContext;

    scrapbook.pages.push(new scrapbookPageModel(scrapbook.pages.length));
};
```

Once again, we're not going to explain this listing in detail. At a high-level, the `onAddTap()` function is called when a user taps the "Add" action bar item, which adds a new scrapbook page to the `pages` observable array.

That's the last of the base code structure for the tablet-specific page. It feels a little anti-climactic, because there's not a lot to show for our work, but stick with us. Laying down this structure upfront will make it easy for us to add the XML markup from the list and details page.

11.2.2 Adding a list to the tablet-specific page

Earlier in this chapter, you updated the UI for the scrapbook list. Let's extract a portion of the existing XML and add it to the tablet-specific page. Listing 11.14 shows the tablet-specific page updated with this XML.

Listing 11.14 Adding the scrapbook list custom UI control to the tablet-specific page

```
<Page loaded="onLoaded">
<Page.actionBar>
    <ActionBar title="Pet Scrapbook" >
        <ActionItem tap="onAddTap" ios.position="right"
                    text="Add" android.position="actionBar"/>
    </ActionBar>
</Page.actionBar>
<GridLayout rows="*" columns="*, 2*">
    <GridLayout rows="*" columns="*"
               items="{{ pages }}"
               itemTap="onItemTap">
        <ListView class="list-group" items="{{ pages }}"
                  itemTemplate="list-item-template">
            <StackLayout orientation="horizontal" class="list-group-item">
                <Image class="thumb img-circle" src="{{ image }}"/>
                <Label class="list-group-item-text"
                      text="{{ title, (title === null ||
                        title === undefined ? 'New' : title + '\s') +
                        ' Scrapbook Page' }}"/>
            </StackLayout>
        </ListView.itemTemplate>
    </ListView>
</GridLayout>
```

```
<GridLayout rows="*" columns="*" col="1">
  <StackLayout>
    </StackLayout>
  </GridLayout>
</GridLayout>
</Page>
```

These changes add the scrapbook list to the tablet-specific page, as seen in figure 11.10. Note the empty list view on the left and the blank space on the right (reserved for the update page).



Figure 11.10 Pet Scrapbook list page shown on a tablet after adding the list view to the left

If you've been following along, go ahead and launch the Pet Scrapbook on a tablet. When you visit the tablet-specific list page and tap the Add button in the action bar area, empty pages will be added to the list in the left. Just in case you aren't following along, you should see something like figure 11.11.

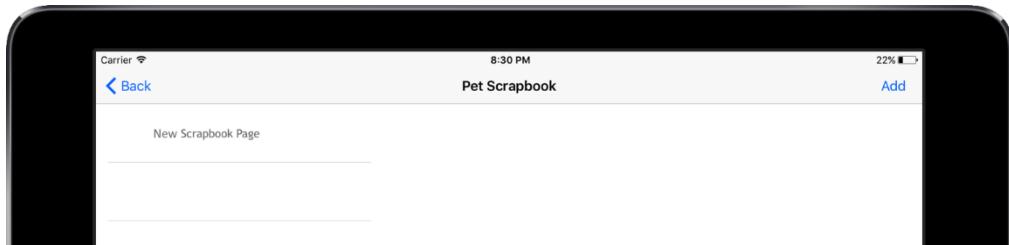


Figure 11.11 Tapping the Add button in the action bar will add a page to the list view on the left.

You'll immediately notice that each button tap adds another page named New Scrapbook Page. Without the update page wired up to the page's binding context, there's no way for you to enter in a page name. Let's finish the tablet-specific view by adding the update page elements.

11.2.3 Adding the update data entry elements to the tablet-specific page

In chapter 10, we created the update page, allowing us to update a scrapbook page. Listing 11.15 shows the contents of this page.

Listing 11.15 The scrapbookUpdate-page.xml file at the end of chapter 10

```

<Page loaded="onLoaded">
  <Page.actionBar>
    ...
  </Page.actionBar>
  <StackLayout>
    <StackLayout class="form"> #B
    ...
    </StackLayout> #B
    <Button class="btn btn-primary" #B
      btn-rounded-sm btn-active" tap="onAddImageTap" #B
      text="Add Image" /> #B
  </StackLayout>
</Page>

```

#A UI code for the action page and data entry form purposely truncated for space

#B These elements will be copied to the tablet-specific page

We've purposefully left out some code within the action bar and the stack layout that contains the data entry form UI, but that technically doesn't matter. What's important is that a subset of this page will be copied to the tablet-specific page. Copy the data entry form stack layout (including its child elements) and the button and place it into the tablet-specific update page. Listing 11.16 shows the tablet-specific page after doing this.

Listing 11.16 The scrapbook-page.land.minWH600.xml file after adding the update page UI elements

```

<Page loaded="onLoaded">
  <Page.actionBar>
    <ActionBar title="Pet Scrapbook" >
      <ActionItem tap="onAddTap" ios.position="right"
        text="Add" android.position="actionBar"/>
    </ActionBar>
  </Page.actionBar>
  <GridLayout rows="*" columns="*, 2*" >
    <GridLayout rows="*" columns="*" >
      <ListView class="list-group" items="{{ pages }}" itemTap="onItemTap">
        <ListView.itemTemplate>
          <StackLayout orientation="horizontal"
            class="{{ isActive ? 'list-group-item active' :
            'list-group-item' }}>
            <Image class="thumb img-circle" src="{{ image }}" />
            <Label class="list-group-item-text"
              text="{{ title, (title === null || title === undefined ?
              'New' : title + '\s') + ' Scrapbook Page' }}"
              style="width: 100%" />
          </StackLayout>
        </ListView.itemTemplate>
      </ListView>
    </GridLayout>
  </GridLayout>

  <GridLayout rows="*" columns="*" col="1">
    <StackLayout>
      <StackLayout class="form"> #A
        <StackLayout class="input-field"> #A
          <Label class="label" text="Name:" /> #A

```

```

<TextField class="input" text="{{ title }}" #A
    hint="Enter a name..." /> #A
</StackLayout> #A
<StackLayout class="input-field"> #A
    <Label class="label" text="{{ birthDate, 'Birth date: ' + #A
        (birthDate === null ? '' : '(' + calcAge(birthDate) + #A
        ' years old)' ) }}" /> #A
    <TextField class="input" editable="false" #A
        text="{{ birthDate, birthDate | #A
            dateConverter(dateFormat) }}" #A
        tap="onBirthDateTap" hint="Enter a birth date..." /> #A
</StackLayout> #A
<StackLayout class="input-field"> #A
    <Label class="label" text="Gender:></Label> #A
    <TextField class="input" editable="false" #A
        text="{{ gender }}" #A
        tap="onGenderTap" hint="Select a gender..." /> #A
</StackLayout> #A
<StackLayout class="input-field"> #A
    <Label class="label" text="Image:" /> #A
    <Image src="{{ image }}" stretch="None" /> #A
    <Label class="footnote" text="{{ (lat, long), #A
        (lat === undefined || long === undefined) ? '' : #A
        'Picture taken at ' + lat + ', ' + long }}" /> #A
</StackLayout> #A
</StackLayout> #A
<Button class="btn btn-primary btn-rounded-sm btn-active" #A
    tap="onAddImageTap" text="Add Image" /> #A
</StackLayout>
</GridLayout>
</GridLayout>
</Page>
#A This is the code we copied from the update page

```

11.2.4 Adding behavior to the update page UI with JavaScript

Now that we've added the update-related UI elements to the tablet-specific page, all that remains is to wire-up the data binding and tap events. At first, you may think wiring up all these components can be done by copying the JavaScript code from the update page. Unfortunately, that's not the case (because the update page assumes the binding context of the page is set to a single scrapbook page). We can reuse quite a bit of the code, but there are a few items we'll need to change. Changing these items at once could be a bit confusing, so let's review each item briefly, then tackle them one at a time:

- Step 1: Data bind the new data-entry from UI elements to the *selectedPage* property of the page's binding content.
- Step 2: When a page is tapped in the left-hand list view, point the *selectedPage* property of the page's binding context to tapped page.
- Step 3: Show modal dialogs when the birth date and gender UI elements are tapped.
- Step 4: Add an image when the Add Image button is tapped.
- Step 5: Save selected page changes to the file system.

STEP 1: CONFIGURING DATA BINDING FOR THE UPDATE UI ELEMENTS WITH THE SELECTEDPAGE PROPERTY

Data binding the update UI elements is a little tricky, because the existing data binding expressions assume the page's binding context points to an observable with properties for the title, birth date, gender, image, latitude, and longitude. But, there's a problem: the tablet-specific page's binding context doesn't have these properties (listing 11.17).

Listing 11.17 The tablet-specific page's binding context

```
var scrapbook = new observable.fromObject({
  pages: new observableArray.ObservableArray(),
  selectedPage: null
});

page.bindingContext = scrapbook;
```

The page's binding context has the *pages* property (which is bound to the list of scrapbook pages on the left-hand side of the page) and the *selectedPage* property (which we intended to point to an index of the *pages* property). Ideally, we'd like the update UI elements we just added to use the *selectedPage* property, as shown in figure 11.12.

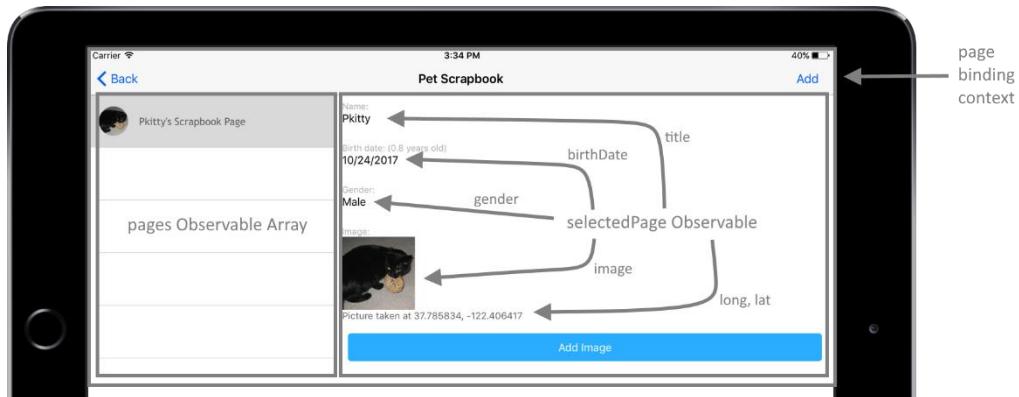


Figure 11.12 The tablet-specific page showing the binding context and how observables should bind to each UI element area

As you'll recall from chapter 8 where you learned about data binding, when the page's binding context is set to an observable, all UI elements within the page gain access to (or inherit) the binding context. Because of this, the list view on the left can bind directly to the *pages* property: `<ListView items="{}{ pages }{}" />`. But, the update UI elements bind directly to properties like *title*: `<TextField text="{}{ title }{}" />`.

We'll tell you how we're going to solve this problem, but take a moment and think about the problem a how you may solve it. As you think, here's a quick tip to get you started. Continue reading after the tip when you're finished thinking.

TIP The page's binding context is inherited to all child elements, and stored in a property called *bindingContext* on each element.

You may have come up with a different solution, but the way we'll solve this problem will help you better understand the process of data binding and the inheritance of the binding context. In the tip, you learned that each UI element has a property named *bindingContext*, which is inherited from its parent element.

Even though we've set the *bindingContext* property on the page only, there's not a reason why we can't set the binding context on another control, breaking the inheritance chain. Let's do just that by getting a reference to the stack layout surrounding the update UI elements in code and setting its binding context to the *selectedPage* observable. Listings 11.18 and 11.19 show the changes to XML and JavaScript code used to get a reference to the surrounding stack layout mentioned above.

Listing 11.18 The update portion of the tablet-specific page with an id added to the stack layout surrounding the data-entry UI elements

```
<GridLayout rows="*" columns="*" col="1">
    <StackLayout id="updateStackLayout" >
        <StackLayout class="form">
            ...
        </StackLayout>
    </StackLayout>
</GridLayout>
#A Additional data-entry UI elements purposefully left out.
```

Listing 11.19 Modified onLoaded handler for the tablet-specific page

```
var observable = require("data/observable");
var observableArray = require("data/observable-array");
var fileSystemService = require("~/data/fileSystemService");
var viewModule = require("ui/core/view"); #A

var page;
var updateStackLayout; #A

exports.onLoaded = function(args) {
    page = args.object;
    updateStackLayout =
        viewModule.getViewById(page, "updateStackLayout"); #A

    var scrapbook = new observable.fromObject({
        pages: new observableArray.ObservableArray(),
        selectedPage: null
    });

    var pages = fileSystemService.fileSystemService.getPages();

    if (pages.length !== 0) {
        pages.forEach(function (item) {
            var model = new scrapbookPageModel();

            model.id = item.id;
```

```

model.title = item.title;
model.gender = item.gender;
model.birthDate = item.birthDate;
model.image = item.image;
model.lat = item.lat;
model.long = item.long;

scrapbook.pages.push(model);
});
}

page.bindingContext = scrapbook;
};

#A A reference to the stack layout is obtained via the view module and saved for later use

```

We've made minor changes to the XML and JavaScript code of the tablet-specific page by adding an *id* field to the stack layout and obtaining a reference to the element with the view module.

Something you probably noticed is that we didn't set the binding context of the stack layout to the selected page. But, why? Look at the *selectedPage* property in the *onLoaded* function: *selectedPage: null*. It's null. If we were to use the *selectedPage* property right away, we'd be setting the stack layout's binding context to null, which effectively does nothing. But, this begs the question, "When should we set stack layout's binding context to the *selectedPage* property?"

Before we move on to the answer, we're calling step 1 complete. Even though we didn't *actually* bind the data-entry by setting the stack layout's binding context to the *selectedPage* property, we setup the structure to allow us to do this later. Let's move on.

STEP 2: HANDLE THE LIST VIEW ITEM TAP EVENT AND SET THE SELECTED PAGE

Before we left the last section, the *selectedPage* property was null and we asked the question, "When should we set stack layout's binding context to the *selectedPage* property?". The answer is to set the binding context when the *selectedPage* property isn't null. But, when isn't the *selectedPage* property null? When a scrapbook page is selected (by tapping an item in the list view on the left of the page).

The series of questions and answers may be a bit long, so let's be pointed: when a list item is tapped, the *selectedPage* property is set to an index within the *pages* observable array. After the selected page is set, the stack layout's binding context should be updated.

Let's look at this in the code. The list view already has an *itemTap* event declared in XML: `<ListView class="list-group" items="{{ pages }}" itemTap="onItemTap">`. Listing 11.20 shows the handling of the event.

Listing 11.20 Handling the itemTap event of the list view on the tablet-specific page

```

exports.onItemTap = function(args) {
    var scrapbook = page.bindingContext;

    scrapbook.set("selectedPage", scrapbook.pages.getItem(args.index));
    updateStackLayout.bindingContext = scrapbook.selectedPage;
};

```

When a list view item is tapped, the index of the tapped item is passed to the event handler through the *index* property. Using this property (and a reference to the page's binding context observable), the

selectedPage property is set to the tapped list item. Then, the magic happens: we set the stack layout's binding context to the selected page, overriding the inherited binding context.

With this change, the tablet-specific page comes to life. If you're following along, go ahead and run the updated app in your emulator. Tapping the Add action bar button adds new pages, and tapping a page in the list view loads the appropriate page on the right. Figure 11.13 shows Pkitty's page selected on top and Riven's page selected on the bottom.

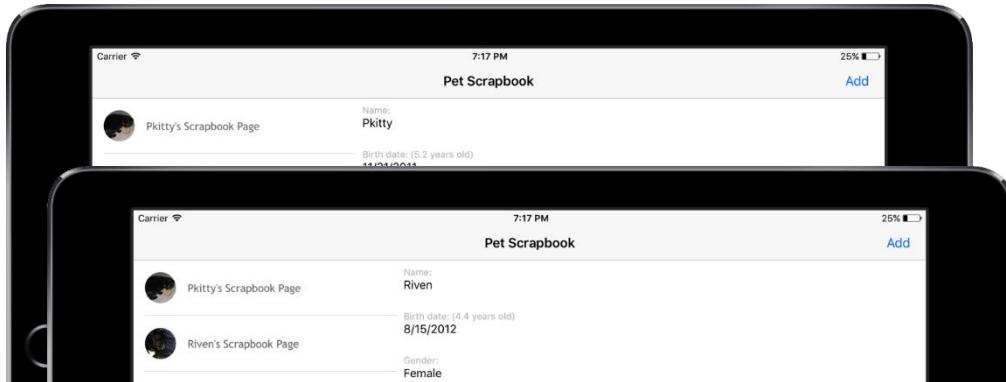


Figure 11.13 The tablet-specific page UI reacting to tapping on different list view items, Pkitty's page selected on top and Riven's page selected on bottom.

The tablet page is coming together; let's move on to the next step.

STEP 3: USING MODALS TO SELECT VALUES FOR THE BIRTH DATE AND GENDER FIELDS

We've already added modals to the Pet Scrapbook earlier in this chapter, and the good news is that the code to implement modals on this page is almost identical. Let's not delay and add handle the tap event on the birth date and gender text fields by copying the code from the details page and adding it to this page. Listing 11.21 shows the code with a few changes that we'll explain.

Listing 11.21 Birth date and gender text field tap event handlers to select values on the tablet-specific page

```
exports.onBirthDateTap = function(args) {
    var modalPageModule = "views/selectDate-page";
    var context =
        { birthDate: updateStackLayout.bindingContext.birthDate }; #A
    var fullscreen = true;

    page.showModal(modalPageModule, context,
        function closeCallback(birthDate) {
            updateStackLayout.bindingContext.set("birthDate", birthDate); #A
        }, fullscreen);
};

exports.onGenderTap = function(args) {
    var modalPageModule = "views/selectGender-page";
```

```
var context = { gender: updateStackLayout.bindingContext.gender }; #A
var fullscreen = true;

page.showModal(modalPageModule, context, function closeCallback(gender) {
    updateStackLayout.bindingContext.set("gender", gender); #A
}, fullscreen);
};

#A References to page.bindingContext were changed to updateStackLayout.bindingContext
```

This code looks remarkably like the code from the details page. In fact, the only thing that changed was the binding context used throughout. The detail page's code used a reference to `page.bindingContext`, because the UI elements on the detail page could use their page's binding context. As you'll recall from earlier in this chapter, the data-entry UI elements were configured to use a different binding context (the `selectedPage` property). Because of this, any references to `page.bindingContext` had to change to `updateStackLayout.bindingContext`.

With these changes, the birth date and gender text fields now open model dialogs to select their values.

STEP 4: HANDLE THE TAP EVENT FOR THE ADD IMAGE BUTTON

The next step is to handle the tap event of the Add Image button. Like the birth date and gender fields, we can copy the code we created previously. Let's grab that code, modify it slightly, and add it to the tablet-specific page (listing 11.22).

Listing 11.22 Birth date and gender text field tap event handlers to select values on the tablet-specific page

```
var camera = require("nativescript-camera");
var geolocation = require("nativescript-geolocation");
var image = require("image-source");

exports.onAddImageTap = function (args) {
    var scrapbookPage = updateStackLayout.bindingContext; #A

    if (!geolocation.isEnabled()) {
        geolocation.enableLocationRequest();
    }

    camera
        .takePicture({ width: 100, height: 100, keepAspectRatio: true })
        .then(function (picture) {
            image.fromAsset(picture).then(function (imageSource) {
                scrapbookPage.set("image", imageSource);
            });
        });

    geolocation
        .getCurrentLocation()
        .then(function (location) {
            scrapbookPage.set("lat", location.latitude);
            scrapbookPage.set("long", location.longitude);
        });
};

#A Reference to page.bindingContext was changed to updateStackLayout.bindingContext
```

Once again, the only change needed is to update the reference to `page.bindingContext` to `updateStackLayout.bindingContext`.

Let's move on to the final step: writing changes to the selected page to the file system.

STEP 5: SAVING SELECTED PAGE CHANGES TO THE FILE SYSTEM

Back in chapter 9, you learned how to use the file system module to save scrapbook pages to the file system. This allowed the Pet Scrapbook to have a permanent record of scrapbook pages that persisted after the app was closed and re-opened. In chapter 10 we added the update page of the Pet Scrapbook, which allowed users to edit the details of a scrapbook page. After editing the details, users could press the Done button, saving the changes to the file system and navigating back to the list of scrapbook pages.

We'd like to reuse code written previously to save changes, so let's revisit the Done button's tap event handler in listing 11.23.

Listing 11.23 The scrapbook update page handling the Done button's tap event

```
exports.onDoneTap = function(args) {
    var scrapbookPage = page.bindingContext;

    fileSystemService.fileSystemService.savePage(scrapbookPage);

    frame.topmost().navigate({
        moduleName: "views/scrapbook-page",
        clearHistory: true,
        transition: {
            name: "slideRight"
        }
    });
};
```

Looking back at the tap event handler on the update page, it saved the updates to the file system by passing the observable referenced by the page's binding context (`scrapbookPage`) to the file system service's `savePage()` function. We can reuse this functionality on the tablet-specific page by passing the `selectedPage` observable to the file system service. Our only problem is there's not a Done button on the tablet-specific page. Let's start by adding a button, then finish by adding code like listing 11.23 to the tablet-specific page. Listing 11.24 updates the UI by adding a Done button next to the Add Image button and wrapping a stack layout around both elements.

Listing 11.24 Done button and stack layout updates to the tablet-specific page

```
<StackLayout orientation="horizontal">
    <Button class="btn btn-primary btn-rounded-sm btn-active"
        tap="onAddImageTap" text="Add Image" />
    <Button class="btn btn-primary btn-rounded-sm btn-active"
        tap="onDoneTap" text="Done" />
</StackLayout>
```

Next, let's add the Done button tap event handler code to the tablet-specific JavaScript file (listing 11.25).

Listing 11.25 The scrapbook update page handling the Done button's tap event

```
exports.onDoneTap = function(args) {
    var selectedPage = updateStackLayout.bindingContext;

    fileSystemService.fileSystemService.savePage(selectedPage);
};
```

After adding this code, the tablet-specific page has the same features as the default pages. Great work!

NOTE We copied and pasted a lot of code in this section, and it feels wasteful. If you're familiar with the *DRY principle* (don't repeat yourself), this section probably made you feel dirty. You may have even been thinking, "there's got to be a better way to share code between the default and tablet-specific pages." Well, there is, and appendix D is dedicated to doing just this: creating re-usable, custom UI controls. Custom UI controls are an advanced topic, and are a bit beyond the scope of this book, but we felt we had to share it with you. So, if you're adventurous or looking for a challenge, check out appendix D.

11.3 Refining the tablet-specific user experience

The tablet-specific page of the Pet Scrapbook is finished, right? Yes, if you're comparing the tablet version to the non-tablet version. Side-by-side both pages have the same features: add a new scrapbook page, update the details of the page, select the birth date and gender using modals, and save the details to the file system. But, we think there's some room for improvement.

Take a minute and use the tablet-specific version and see if you feel there's room for improvement. If you find anything, build a list, and compare it to the list we'll share with you.

When we sat down and used the Pet Scrapbook for a few minutes, several questions were raised in our mind:

- When a page is selected from the list, how do we know which page was selected?
- When a page is added, wouldn't it be nice if the app selected that page automatically?
- When the page loads for the first time, why do blank data-entry UI elements show on the right?
- Why do the data-entry UI elements show after the Done button is pressed?

Our list of questions is short, appear to be quickly remedied, and are high-value (meaning they can significantly improve the app). You may have noticed some of the same short-comings of the app (plus one or two more), so let's work through correcting each of these items together.

11.3.1 Visual feedback for the selected page

This first issue we'll address is giving users visual feedback when a page is selected from the list of scrapbook pages. Ideally, it would be nice to highlight a list item when it's been tapped. When a different item is tapped, it would be highlighted. Because styling a list view item differently when selected is a common occurrence, the core theme package has a special CSS class combination (`list-group-item active`)

that can be applied to a selected list item. Let's use this style and add a conditional styling rule to the list view item template by using a data binding expression (listing 11.26).

Listing 11.26 The scrapbook update page handling the Done button's tap event

```
<ListView class="list-group" items="{{ pages }}" itemTap="onItemTap">
  <ListView.itemTemplate>
    <StackLayout orientation="horizontal"
      class="{{ isActive ? 'list-group-item active' : #A
      'list-group-item' }}> #A
      <Image class="thumb img-circle" src="{{ image }}" />
      <Label class="list-group-item-text"
        text="{{ title, (title === null || title === undefined ?
          'New' : title + '\'s') + ' Scrapbook Page' }}"
        style="width: 100%" />
    </StackLayout>
  </ListView.itemTemplate>
</ListView>
#A When the isActive property is true, style the list item with the active class
```

By adding a data binding expression to the class property of the stack layout, we're able to control when a list item is styled with the *list-group-item* class (not selected) or the *list-group-item* and *active* classes (selected). But, for this to work, the *isActive* property needs added to the scrapbook page observable. Listing 11.27 shows the change made to the `scrapbookPageModel()` function, which returns the observable.

Listing 11.27 The scrapbookPageModel() function updated to include the isActive property

```
function scrapbookPageModel(id) {
  var model = new observable.Observable({
    id,
    title: null,
    birthDate: null,
    gender: null,
    image: null,
    lat: null,
    long: null,
    isActive: false #A
  });

  model.calcAge = function(birthDate){
    var now = Date.now();
    var diff = Math.abs(now - birthDate) / 1000 / 31536000;

    return diff.toFixed(1);
  };

  return model;
}
#A By default, a scrapbook page shouldn't be active (or selected)
```

When a new page is created, it isn't selected or active, so the *isActive* property is set to false.

Now that we've added the *isActive* property, the only thing left to do is to set it to true when a list item is selected, and reset its value to false when another item is selected. Let's update the *itemTap* event handler's code to do just this (listing 11.28).

Listing 11.28 Updated itemTap event handler on the tablet-specific page, incorporating the isActive observable property

```

function resetActivePage() {
    var scrapbook = page.bindingContext;

    scrapbook.pages.forEach(function (item) { #A
        item.set("isActive", false);           #A
    });
}

scrapbook.selectedPage.set("isActive", true); #B
}

exports.onItemTap = function(args) {
    var scrapbook = page.bindingContext;

    scrapbook.set("selectedPage", scrapbook.pages.getItem(args.index));
    updateStackLayout.bindingContext = scrapbook.selectedPage;

    resetActivePage();
};

#A Reset all pages isActive property to not be selected
#B Update the selected list item to show as selected

```

The `resetActivePage()` function handles all of the heavy lifting. First, we loop through the page's observable array and reset all the `isActive` properties to be false (not selected). Then, when the `selectedPage` property and the data-entry form's binding context are set to the selected page, we set the `isActive` property of the selected page to true. Let's check it out in the emulator (figure 11.14).

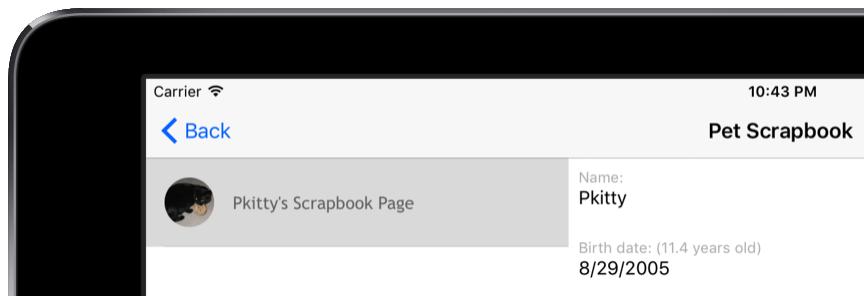


Figure 11.14 The tablet-specific version of the Pet Scrapbook showing which page is selected

Wow! That change made a huge difference: now it's perfectly clear which page is selected. Let's move on to the next issue.

11.3.2 Auto-selecting a page when it's added

The next issue is a natural extension of the previous. Now that we know when a page is selected, wouldn't it make sense to auto-select a page as soon as it's added? After all, when you add a page, the first thing you'll want to do is update its information, so why not save the user the extra tap and auto-select the newly-added page?

Let's update the action bar Add button's tap event handler, as shown in listing 11.29.

Listing 11.29 Updated itemTap event handler on the tablet-specific page, incorporating the isActive observable property

```
exports.onAddTap = function(args) {
    var scrapbook = page.bindingContext;

    scrapbook.pages.push(new scrapbookPageModel(scrapbook.pages.length));

    scrapbook.set("selectedPage",
        scrapbook.pages.getItem(scrapbook.pages.length - 1));
    resetActivePage();
};
```

The changes to the Add button's tap event handler are straightforward. First, we set the selected page to the last index of the page's observable array (because the new scrapbook page is always added to the end of the array). Then, we reset the active page using the function we defined previously. It's that simple. Now, when we add a new page, it's automatically selected. Figure 11.15 shows the newly added page selected automatically.

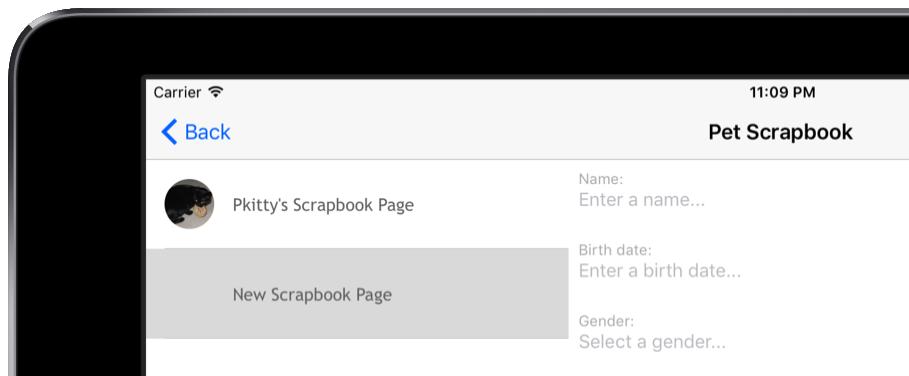


Figure 11.15 The tablet-specific page auto-selecting a scrapbook page when it's added, saving users an extra tap.

11.3.3 Hiding the data-entry UI elements when the page loads

Although this issue may not seem important, it's a not-so-secret pet peeve of ours.

TIP Don't show UI elements, especially data-entry elements, when they're not applicable.

Yeah, believe it or not, we run across UIs that display data-entry elements, even when they're not applicable. We feel this is nothing more than egregious, so let's fix the app by hiding the data entry UI elements until a scrapbook page is selected.

TIP To control the visibility of a UI element, use a data-binding expression in the visibility property of a UI element. Setting the visibility property to collapsed hides the element (and its child elements). To show the UI element, set the visibility property to visible.

Listing 11.30 shows the update to the grid layout on the right of the tablet-specific page that keeps the grid layout and its children hidden until the `selectedPage` property is no longer set to `null`.

Listing 11.30 Controlling the visibility of the grid layout and it's child UI elements with the visibility property and a data-binding expression.

```
<GridLayout rows="*" columns="*" col="1"
    visibility="{{ selectedPage === null ? 'collapsed' : 'visible' }}">
    <StackLayout id="updateStackLayout">
        ...
        #A
    </StackLayout>
</GridLayout>
#A Child elements purposefully left out to save space
```

Again, the changes to correct this are straightforward: the data-binding expression for the visibility property inspects the `selectedPage` property. When its value is `null`, the grid layout and its child UI elements are hidden, and when a page is selected, the grid layout is shown. Figure 11.16 shows the updated UI before and after selecting a scrapbook page from the list view.

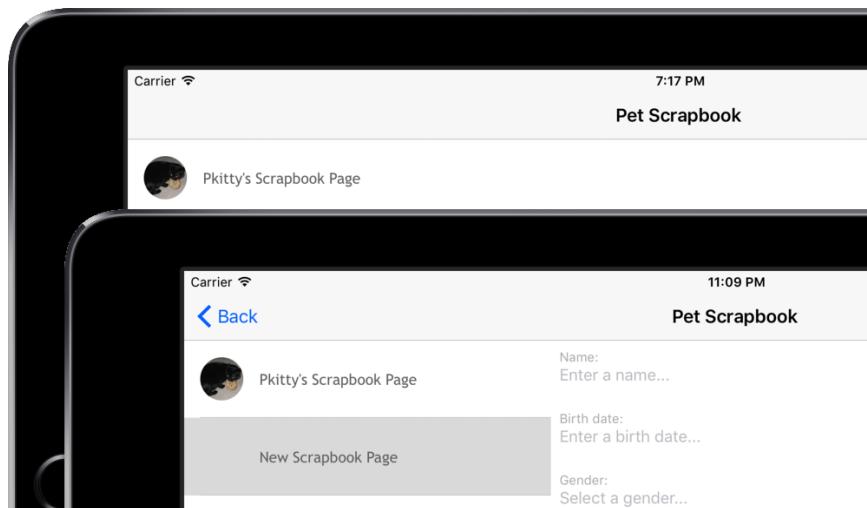


Figure 11.16 The tablet-specific page hiding the data-entry UI elements before selecting a list item (top), and afterwards (bottom) showing the data-entry UI elements

11.3.4 Hiding the data-entry UI elements after pressing Done

Now that we have a mechanism for controlling when the data-entry UI elements are displayed, we can be more judicious. The current functionality of the page can be a bit confusing. In fact, when you press the

Done button, nothing happens. Technically, the scrapbook page is saved to the file system, but you receive no feedback from the UI.

Let's fix that and hide the data-entry UI after pressing the Done button. Listing 11.31 shows the updated tap event handler.

Listing 11.31 Hiding the data-entry UI elements after pressing the Done button

```
function resetActivePage() {
    var scrapbook = page.bindingContext;

    scrapbook.pages.forEach(function (item) {
        item.set("isActive", false);
    });

    if (scrapbook.selectedPage != null) {
        scrapbook.selectedPage.set("isActive", true);
    }
}

exports.onDoneTap = function(args) {
    var scrapbook = page.bindingContext;

    fileSystemService.fileSystemService
        .savePage(scrapbook.selectedPage); // #A

    scrapbook.set("selectedPage", null); // #B
    resetActivePage(); // #B
};

#A Save the changes to the file system
#B Clear the selectedPage observable and ensure none of the pages in the list view is selected
```

Compared to the other issues we've addressed, there's a little more work to do here, but it's still relatively limited. First, we've updated the `resetActivePage()` function to check for a selected page value of `null`. Then, when the Done button is tapped, we set the selected page to `null` and call the `resetActivePage()` function to adjust the list view's selected item.

And with that final change, we're going to call the Pet Scrapbook complete. The most recent changes were minor but make the app much more user-friendly, demonstrating the value of our attention to detail.

A final version of the code from this chapter can be found at <https://github.com/mikebranstein/TheNativeScriptBook/tree/master/Chapter11>.

11.4 Summary

In this chapter, you learned the following:

- Modal pages are defined with XML, JavaScript, and CSS (just like regular pages), but you use the `showModal()` function and a close callback function to open and close a modal.
- To send data from a modal dialog to the previous page, you pass it back as an argument of the close callback function.
- UI elements inherit their binding context from their parent UI control, but can be overridden by setting their `bindingContext` property directly.

- UI element visibility can be toggled by setting the *visibility* property of an element to either *collapsed* or *visible*.

11.5 Exercises

In the final section of this chapter, we shared a list of four issues we felt the Pet Scrapbook app had. We also asked you to write down a list of issues you identified. Using the lists developed, complete the following exercises.

- The `resetActivePage()` function is inefficient because it loops through all of the page's observable array to reset each item's *isActive* property to false. Assuming that only one scrapbook page can have its *isActive* property set to true, developer a more efficient solution.
- The title and done button of the update page feel better as an action bar title and action button. Move them into an action bar at the top of the update page.
- Challenge: Using the list of issues you developed, correct one of the issues you identified.

11.6 Solutions

A more efficient `resetActivePage()` function is included in listing 11.32. In summary, the more efficient solution keeps track of the previously-selected page in a global variable.

Listing 11.32 More efficient `resetActivePage()` function

```
function resetActivePage() {
    var scrapbook = page.bindingContext;

    previouslySelectedPage.set("isActive", false); #A

    if (scrapbook.selectedPage != null) {
        scrapbook.selectedPage.set("isActive", true);
    }
}
#A Assumes that prior to setting the selectedPage, other code blocks keep track of
the previously selected page
```

To move the update page title and done button to an action bar, remove the title and done button elements and add the UI elements from listing 11.33 to the top of the `scrapbookUpdate-page.xml` file.

Listing 11.33 Action bar code for a title and done button

```
<Page actionBar>
    <ActionBar title="{{ title, (title === null || title === undefined || title === ''
? 'New Page' : title + ' Page') }}">
        <ActionItem tap="onDoneTap" ios.position="right"
            text="Done" android.position="actionBar"/>
    </ActionBar>
</Page actionBar>
```


12

Deploying an Android app

This chapter covers

- How to prepare your app to be deployed
- How to deploy your app to the Google Play store

In the last several chapters, we've continued to refine the Pet Scrapbook app while teaching you about supporting multiple screen resolutions, using modals, and styling apps with themes. Now that we've finished adding features to the Pet Scrapbook, what's next? Writing an app solely for the app's sake isn't our motivation: we aspire to share our work. And the best way to share is by publishing the Pet Scrapbook to the app stores.

Over the next three chapters, you'll learn how to finalize apps and prepare them for the app stores. In this chapter, we'll cover how to get ready for the Google Play store by creating app icons, a launch screen, and configuring the app for various devices. In the following two chapters, we'll cover the same learning points, but focus on Apple's App Store.

Let's get started!

12.1 Customizing Android apps with the App_Resources folder

Let's take a brief journey back to chapter 3, where you learned about the structure of a NativeScript app. You'll recall the *App_Resources* folder is a special folder holding platform-specific customizations. When we first introduced this to you, we said we'd come back to it later. That time has come.

Customizations (like app icons and launch screens) are done by updating files in the *App_Resources* folder. There's a lot going on under this folder, so let's take a closer look (figure 12.1).

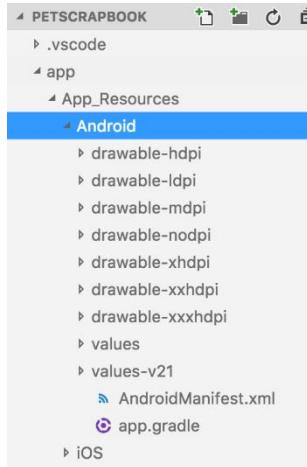


Figure 12.1 The App_Resources/Android folder contains files and folders that can be updated to configure Android-specific app settings.

Figure 12.1 shows two subfolders beneath the *App_Resources* folder: *Android* and *iOS*. These two folders house platform-specific files for *Android* and *iOS* respectively. In this chapter, we'll be focusing on the *Android* folder. If you're interested in learning about the *iOS* folder, check out the next two chapters.

12.1.1 App_Resources/Android contents

The first series of folders you'll notice follow the naming convention of *drawable-**. The *drawable* folders are used by *Android* to store different resolutions of all the image resources in the Pet Scrapbook so they look correct on devices that support different resolutions and DPI. If you recall, we used these in chapter 7 to add image resources for the Tekmo app. To help jog your memory, table 12.1 details how each *drawable* folder maps to various device DPIs.

Table 12.1 Approximate Android device DPI densities and the corresponding App_Resources folder

Size	App_Resources Folder	Approximate DPI
low	drawable-ldpi	~120 DPI
medium	drawable-mdpi	~160
high	drawable-hdpi	~240
extra-high	drawable-xhdpi	~320
extra-extra-high	drawable-xxhdpi	~480
extra-extra-extra-high	drawable-xxxhdpi	~640

The next two folders are the *values* folders. These folders contain XML files that define the default look and feel of Android themes (figure 12.2).

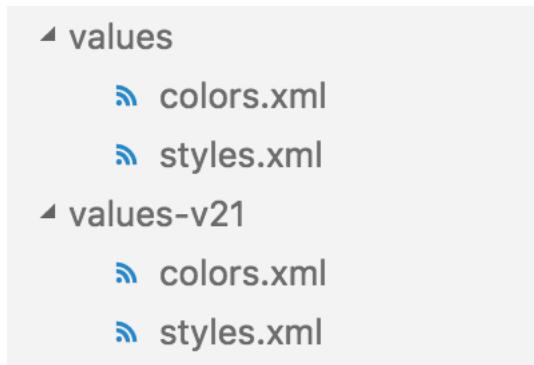


Figure 12.2 The default contents of the *values* folders in the Pet Scrapbook app.

There are two folders because they apply to different versions of Android. You don't need to know the details beyond the fact that devices running an Android version of 5.0 and higher refer to the *values-v21* folder, and older versions refer to the *values* folder.

NOTE The v21 folder extension refers to the Android API and SDK version 21. Android devices running Android v5.0 use API version 21. This may sound confusing, but as a NativeScript developer, you really don't need to know the details.

Looking close at figure 12.2, the *colors.xml* and *styles.xml* files define default colors and styles for Android apps. You're welcome to open the files and inspect them further: it's nice to know the files are there, but we won't be customizing them. We'll come back to the *values* folders a little later in this chapter, so don't forget about them.

NOTE If you want to learn more about the themes, colors, and styles for an Android app, you can visit <https://developer.android.com/guide/topics/ui/themes.html>.

The last files in the *App_Resources/Android* folder are the *app.gradle* and *AndroidManifest.xml* files. The *app.gradle* file is used to define custom build settings for an Android app, which is used by the NativeScript CLI to create transform your app's source code into a complied Android app. We won't be going into detail about the *app.gradle* file, but if you want to learn more, go to <https://developer.android.com/studio/build/build-variants.html>. The *AndroidManifest.xml* file is like a configuration file for the Pet Scrapbook app, and is the most significant of the files in the *Android* folder. We'll be using this file extensively to make modifications to the Pet Scrapbook.

Let's take a closer look at the *AndroidManifest.xml* file.

12.2 *AndroidManifest.xml* customizations

Every NativeScript app that is being deployed to Android must have an application manifest file named *AndroidManifest.xml*. This file is generated when you target android as a platform using the `tns platform add android` CLI command.

DEFINITION The *AndroidManifest.xml* file is an Android system file that provides essential information about your app to the Android runtime such as app version, supported screen configurations, app icon, and more.

The *AndroidManifest.xml* file is where we will configure the app icons for the Pet Scrapbook app. This file must exist in the root of the *App_Resources/Android* folder, as shown in figure 12.3.

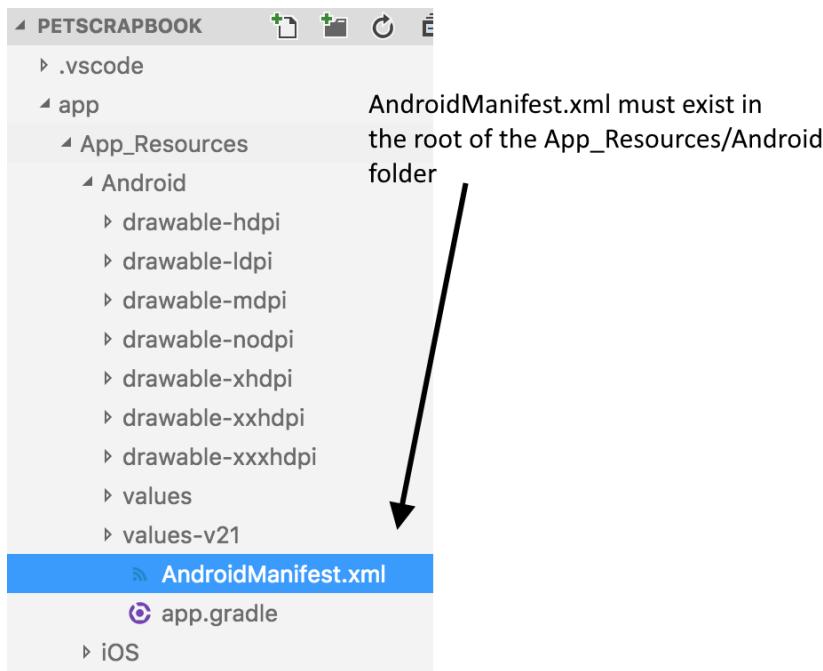


Figure 12.3 The location of the *AndroidManifest.xml* file within the Pet Scrapbook project (and all NativeScript projects).

WARNING Don't move or rename the *AndroidManifest.xml* file. If you do, your app won't run. The *AndroidManifest.xml* file is officially defined by the Android API and not by NativeScript (NativeScript simply provides a default for you). Visit <https://developer.android.com/guide/topics/manifest/manifest-intro.html> for full documentation of the *AndroidManifest.xml* file.

An item defined in the *AndroidManifest.xml* file is the app icon for the Pet Scrapbook app. Let's look at how we can create an app icon and update it for the Pet Scrapbook app.

12.2.1 App icons

All apps need an app icon or they can't run properly on a device. But that's not all an app icon is. App icons are the first thing that a potential user sees about your app. The app icon should convey meaning and be easily identifiable or memorable so that users can quickly find your app. Chances are good that you recognize at least one of the app icons in figure 12.4. These app icons are quickly distinguishable because they are using company logos.

TIP Getting your app icon right is incredibly important. It should be simple and representative of your app. Most of all, it should be artistically attractive. If you're not artistic, that's ok. Pair up with a friend.

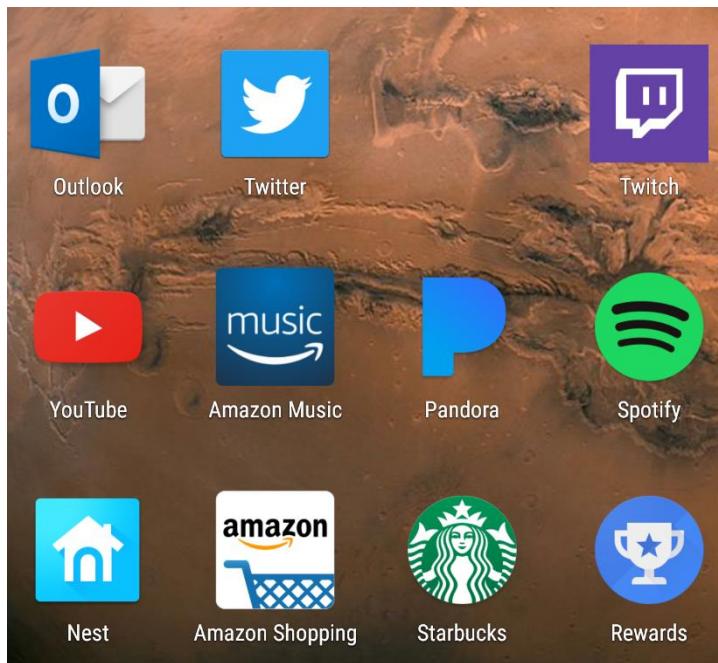


Figure 12.4 Some common app icons that you may recognize.

When we scaffolded the Pet Scrapbook with the `tns create PetScrapbook` CLI command, a default app icon was created (figure 12.5).



Figure 12.5 The default NativeScript app icon, created by the CLI.

You might be thinking that the NativeScript CLI creates a single app icon, but that's not the case. If you recall from chapter 7, NativeScript apps can run on a variety of devices, each with varying screen dimensions and resolutions. To accommodate the various devices, the CLI creates multiple app icons. When the icons are created, they are stored in the various drawable folders inside the `App_Resources/Android` folder, as shown in figure 12.6.

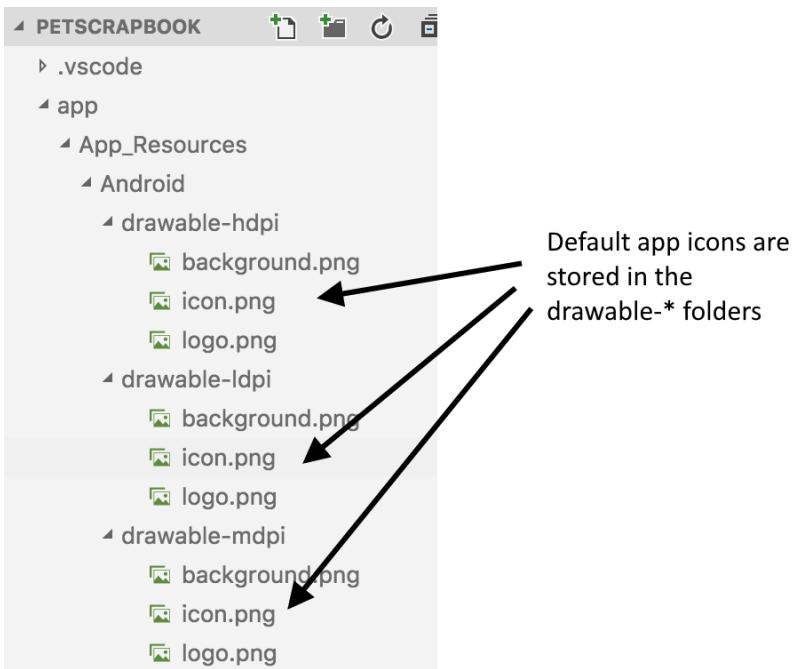


Figure 12.6 The default app icons of the Pet Scrapbook are named `icon.png` and stored inside the various drawable folders.

The default app icons created by the CLI are named icon.png and located in the *drawable-** folders.

DEFINITION The drawable folders are specific system folders defined by the Android API that is used to store bitmap graphic files or XML. In this book, we will be discussing bitmap graphics. If you would like to learn more about XML files in the drawable folder, please visit the official Android documentation at <https://developer.android.com/guide/topics/resources/drawable-resource.html>.

This isn't the first time we've talked about the *drawable-** folders. Back in chapter 7, you learned how you could use the *drawable-** folders to hold DPI-specific image resources. Do you see the parallel between images, app icons, and the *drawable-** folders?

NOTE Wait for it...click! There it was. App icons are images, and Android uses the *drawable-** folders to display DPI-specific images on various devices.

Now that you know app icons are treated just like image resources, let's take a closer look at the *drawable-** folders.

APP ICON DPIS

From our earlier discussions of the *drawable-** folders in chapter 7, you might recall that these folders follow a specific naming convention. The folders start with the *drawable-* naming convention, followed by a DPI code (corresponding to a device's DPI). There's a lot of DPIS to remember from chapter 7, so we've included table 12.2 for reference. It shows a full listing of the different drawable folders and device DPIS supported by Android.

TIP A listing of all the available drawable folders can be found at <https://developer.android.com/guide/topics/resources/providing-resources.html#AlternativeResources>.

Table 12.2 Drawable folder names and corresponding device DPIS.

Folder name	Device DPI
drawable-nodpi	all devices
drawable-ldpi	120
drawable-mdpi	160
drawable-hdpi	240
drawable-xdpi	320
drawable-xxdpi	480
drawable-xxxdpi	640

Wow, that's a lot of DPIs! When creating an app, you need to create seven different app icons—one for each `drawable-*` folder! If you moonlight as a graphic artist, then creating the various app icons won't be a problem for you, but if you're like us, you'll need some help. Luckily there are a few shortcuts. Let's explore these shortcuts and create an app icon for the Pet Scrapbook.

12.2.2 Customizing app icons

Before we get started, we'll need an app icon for the Pet Scrapbook. We're not artistically inclined, so we partnered with an artistic friend, Batman.

NOTE Yeah, his name is Batman. Because he's awesome. Special thanks to you, Batman!

With Batman's help, we developed an app icon for the Pet Scrapbook (figure 12.7).



Figure 12.7 The Pet Scrapbook app icon, as created by our friend, Batman.

Let's download a copy of the app icon before we continue. You can get a copy at <https://github.com/mikebranstein/TheNativeScriptBook/blob/master/PetScrapBook.png>.

APP ICON SHORTCUTS

To customize the app icon of the Pet Scrapbook, remember that we need to create seven versions of the image shown in figure 12.7. We could open our favorite image editing program, change the image size, and re-save each resized image, but that is a ton of work. Instead, we're going to take a shortcut and use our NativeScript image builder at <http://nsimage.brosteins.com>.

TIP To save time, use an automatic app icon resizer, like the Brostein's NativeScript image builder at <http://nsimage.brosteins.com>.

Figure 12.8 shows our NativeScript image builder tool that you can use for generating app icons.

NOTE The NativeScript Image Builder tool is a free to use tool that you can choose to use to help scale all your NativeScript app images appropriately. If you choose not to use the tool, then you will need to scale your images appropriately so they look correct on the different device DPIs shown in Table 12.1.

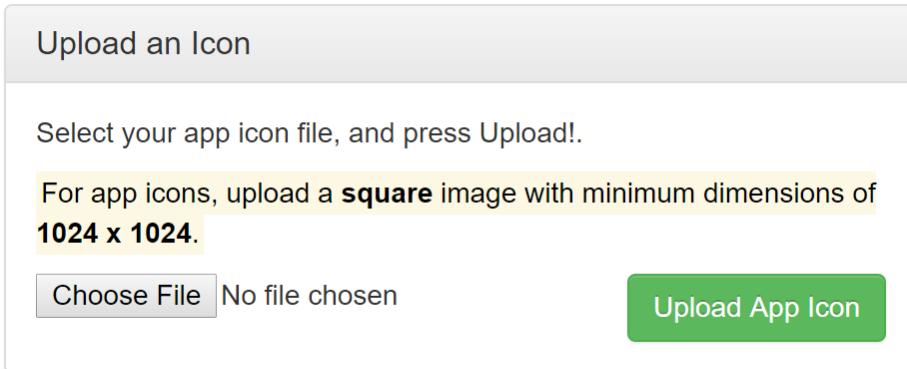


Figure 12.8 The NativeScript image builder tool showing how to upload an icon file to get back scaled icon resources to use in the Pet Scrapbook.

You may recall using our image builder tool back in chapter 7, where we uploaded image resources and resized them for various screen resolutions.

NOTE Our free tool also allows you to upload regular image resources that you would like to generate scaled resources for your NativeScript app.

The image builder tool also helps you resize app icons. Let's upload the Pet Scrapbook app icon. Use the *Choose File* button, locate the Pet Scrapbook icon, and upload it with the *Upload App Icon* button.

WARNING It's important to note that app icons should be square and a minimum of 1024 x 1024 pixels. Our image builder warns and won't let you upload images of smaller dimensions that aren't square, so you don't have to worry about uploading an invalid file. Regardless, it's nice to know ahead of time.

After uploading the Pet Scrapbook app icon, you receive a zip file with images that are scaled for the different device DPIs.

NOTE Our image builder created app icons for both Android and iOS. Android app icons are in the Android folder. iOS app icons are in the iOS folder.

Figure 12.9 shows the contents of the Android folder.

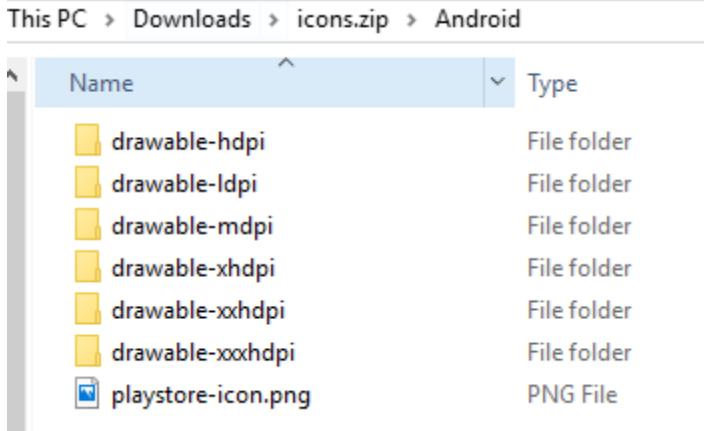


Figure 12.9 The contents of the icons.zip file that our image builder tool returns after you upload the Pet Scrapbook icon file.

The zip file that you get back from the NativeScript Image Builder contains icon.png files inside the various *drawable-** folders. Extract the icon files and update the Pet Scrapbook project by placing them in the appropriate folder.

That's it!

When the Pet Scrapbook is installed on an Android device, the appropriately sized app icon will be used and displayed. Figure 12.10 shows the resulting Pet Scrapbook icon installed on an Android device.

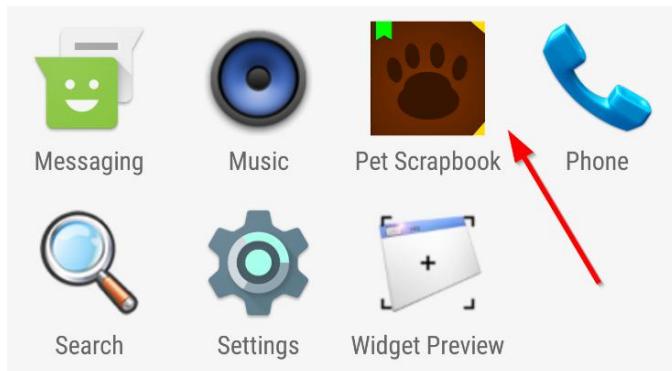


Figure 12.10 The Pet Scrapbook installed to an Android device, showing the updated app icon.

TIP When you change files and folders within the *App_Resources* folder, you'll need to re-build your app using the `tns build android` CLI command. If you notice that your app icon doesn't change,

make sure you run `tns build android`. If your app icon doesn't change after running `tns build android`, try running `tns platform remove android`, then `tns run android`.

Now that we've seen the Pet Scrapbook icon updated, let's take a moment to understand how NativeScript knows to use the `icon.png` files in the `drawable-*` folders.

12.2.3 Understanding app icons and the `AndroidManifest.xml` file

Earlier in this chapter, you learned that the `AndroidManifest.xml` file was used to configure Android apps, including settings such as the app icons. Let's look at the file closer, focusing on the application XML node.

Listing 12.1 The `AndroidManifest.xml` application node showing the app icon setting

```
<application
    android:name="com.tns.NativeScriptApplication"
    android:allowBackup="true"
    android:icon="@drawable/icon" // #A
    android:label="@string/app_name"
    android:theme="@style/AppTheme" >
    <activity
        android:name="com.tns.NativeScriptActivity"
        android:label="@string/title_activity_kimera"
        android:configChanges="keyboardHidden|orientation|screenSize">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <activity android:name="com.tns.ErrorReportActivity"/>
</application>
#A The name of the app icon in the drawable folders, telling the Android app which image file to use as an app icon
```

Listing 12.1 shows the application node of the `AndroidManifest.xml` file. Don't worry. You don't need to know the details of this file, but it's nice to know what's going on behind the scenes.

TIP Even though you don't need to know the details of the `AndroidManifest.xml` file (and the `application` element of the file), you might be interested in learning more. For more information about the `application` element, check out <https://developer.android.com/guide/topics/manifest/application-element.html>.

Listing 12.1 points out the `android:icon="@drawable/icon"` attribute, which tells Android which image file to use as the app icon. `@drawable` refers to the `drawable-*` folder that appropriate matches the screen resolution of a device, and `icon` is the file name (without extension) of the app icon image file.

NOTE You may be wondering why the icon image file is referenced without an extension. Because the image is located across the various `drawable-*` folders, it's treated as an image resource. When

referencing image resources, you don't need a file extension. If you'd like a deeper explanation, look back at chapter 7.

NOTE If you are interested in the finer details of how Android selects the best match of a drawable, please refer to <https://developer.android.com/guide/topics/resources/providing-resources.html#BestMatch>.

Something nice about NativeScript (and using our image builder tool) is that the app icon configuration in `AndroidManifest.xml` and the images downloaded from our tool are already configured with the appropriate app icon names. Because of this, when you replace the `icon.png` files in the `drawable-*` folders, your app icon will be updated accordingly.

ANOTHER APP ICON SHORTCUT, BUT DON'T DO IT!

Even though it's a good practice to supply app icons for all of the various screen resolutions, you need to supply a single app icon only.

If you have a limited amount of time, you may want to create a single app icon and place it in the `drawable-nodpi` folder. When you place an app icon in the `drawable-nodpi` folder and leave it out of the other `drawable-*` folders, Android will use the app icon from the `drawable-nodpi` folder.

WARNING Don't do this. Even though it may feel like a shortcut, using a single app icon may create a suboptimal user experience. Take the extra minute and use our image builder tool to get all of your app icons.

Now that we've updated the Pet Scrapbook app icon, let's create a launch screen.

12.3 Launch Screens

All Android apps must have launch screens (sometimes called splash screens).

DEFINITION A launch screen is a graphical UI, usually displaying an image or logo, that displays while the app is loading.

The purpose of the launch screen in Android apps is to set the stage and introduce the app to the user. In the world of Android, an app's launch screen can be creative, but at the same time simple. Furthermore, a launch screen should adhere to Google's Material Design launch screen guidelines.

DEFINITION Google's Material Design launch screen guidelines are a set of rules making recommendations on Android-specific UI styles for launch screens. For more details on Google's Material Design launch screen guidelines, check out <https://material.google.com/patterns/launch-screens.html>.

In Android apps, there are two types of launch screens: *placeholder UI* and *branded* launch screens. The two types of launch screens are similar, and you really don't need to know the specific differences. But, at a high-level, placeholder UIs are more simplistic, showing a minimal status bar while the app

loads. Branded launch screens, are more complex, and give you an opportunity to highlight your app's brand with additional visual UI elements.

TIP If you're having trouble deciding which launch screen to use, defer to the branded launch screen. In our opinion, placeholder UIs are blah, a technical term for boring. Spending an extra 5 minutes to configure a branded UI launch screen will pay off. NativeScript apps are also pre-configured to use a branded launch screen.

The Pet Scrapbook app will use a branded launch screen, meaning that we'll be highlighting the Pet Scrapbook brand by displaying the Pet Scrapbook logo.

EXPLORING THE BRANDED LAUNCH

Just like app icons, the NativeScript CLI creates a default branded launch screen for apps when the app is scaffolded using the `tns create` CLI command. The launch screen is configured through the `splash_screen.xml` file in the `App_Resources/Android/drawable-nodpi` folder (figure 12.11).

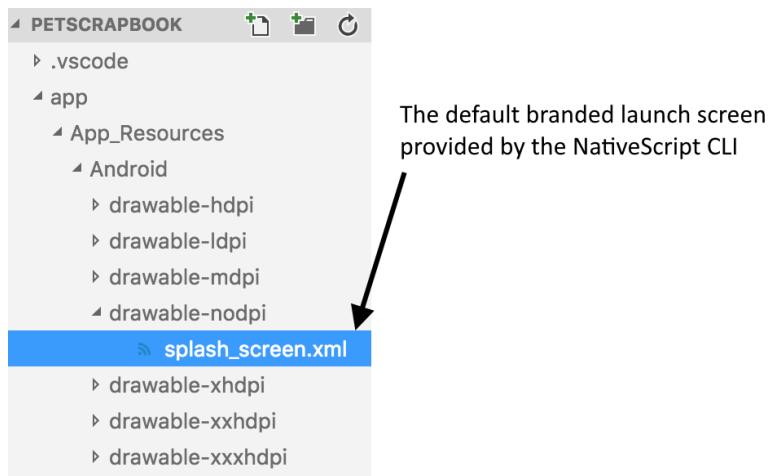


Figure 12.11 The default launch screen is created by the NativeScript CLI and stored in the `drawable-nodpi` folder.

Now that we know where the launch screen is stored, let's look a bit closer and customize it to include the Pet Scrapbook logo.

12.3.1 Updating the launch screen

The default launch screen is different from other pages you've created in NativeScript: instead of using NativeScript UI elements, it uses native Android components. Unfortunately, this is one spot where we will not be able to use the NativeScript XML syntax for creating a page because when the launch screen is being loaded by the Android runtime before the NativeScript virtual machine runs your app code.

NOTE If you don't know native Android UI components, don't worry. Customizing the Android launch screen is incredibly easy (all you need to do is replace two image files). Stick with us, and you'll have nothing to worry about.

Let's start by looking at the default launch screen provided by the NativeScript CLI. Listing 12.2 shows the `splash_screen.xml` file, where the launch screen is defined.

Listing 12.2 The `splash_screen.xml` file, which defines a NativeScript app's launch screen

```
<layer-list xmlns:android="http://schemas.android.com/apk/res/android"  
    android:gravity="fill">  
    <item>  
        <bitmap android:gravity="fill" android:src="@drawable/background" /> // #A  
    </item>  
    <item>  
        <bitmap android:gravity="center" android:src="@drawable/logo" /> // #B  
    </item>  
</layer-list>  
#A Points to background.png, an image filling the background of the launch screen  
#B Points to logo.png, the image centered on the launch screen
```

There's not much going on in listing 12.2, because it's only eight lines long, and even if you don't know the Android UI markup language, it's evident that the screen is composed of two image references: `@drawable/background` and `@drawable/logo`.

The `@drawable` convention is the same convention you learned about earlier in this chapter. When an Android app loads, it looks for a *drawable-** folder (depending on its screen resolution) and loads the appropriate *background.png* and *logo.png* image files. The *background.png* image is expanded to fill the background of the launch screen. The *logo.png* image is centered on top of the background.

If you inspect the various `drawable-*` folders in the `App_Resources/Android` folder, you'll find `background.png` and `logo.png` files for each resolution (figure 12.12).

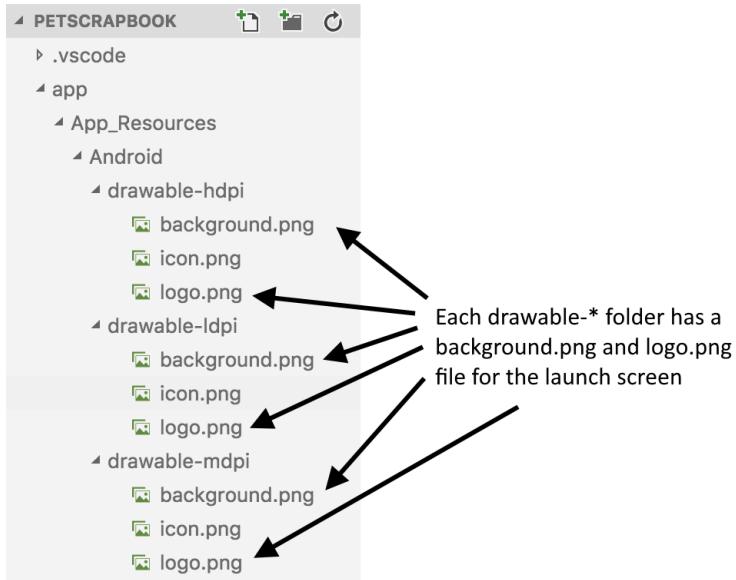


Figure 12.12 The drawable-* folders containing background.png and logo.png files, used to create a launch screen for various screen resolutions.

Now that you know how an app's launch screen is created in NativeScript, let's look at the default launch screen when an Android app loads (figure 12.3).

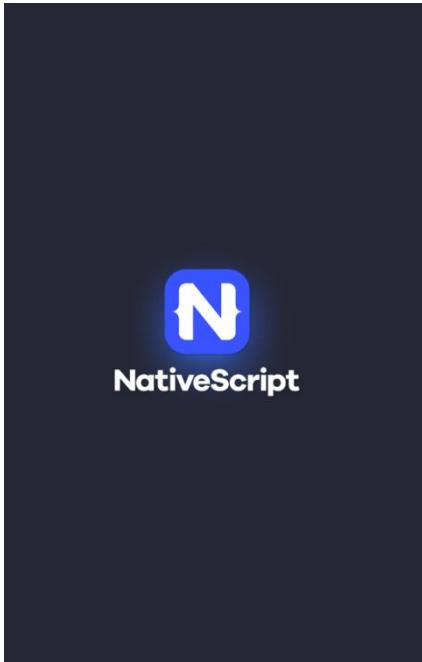


Figure 12.13 The default launch screen of a NativeScript app.

You've probably seen this launch screen dozens of times if you've been following along in the book. Let's change it for the Pet Scrapbook.

REPLACING THE BACKGROUND.PNG TO UPDATE THE LAUNCH SCREEN

Updating the launch screen is like updating app icons: change the image resources in the drawable-* folders. More specifically, we'll replace the *background.png* and *logo.png* files.

Because we'll have to create images for the various screen resolutions supported by Android, we'll be using our image builder website at <http://nsimage.brosteins.com>.

Let's start with the background image. Choose a color that is complimentary to the Pet Scrapbook logo and create a high-resolution (3000 x 4000 pixels) image of that color. Name the image *background.png*. Alternatively, you can download an image we've already created from <https://github.com/mikebranstein/TheNativeScriptBook/blob/master/background.png>.

TIP Background images with a solid color work best because the image is stretched to fit a device's screen resolution and dimensions. Sticking with a solid color not only ensures the background will look right, but it also creates a clean-looking launch screen.

Using the image upload feature of our NativeScript image builder, selected *Static Image – Android*, click the *Choose File* button, locate the *background.png* image, and press *Upload App Image* (figure 12.14).

NOTE Make sure you're using the image upload feature on the left side of our image builder site. The left side is for static images (like the background image). The right side is for app icons.

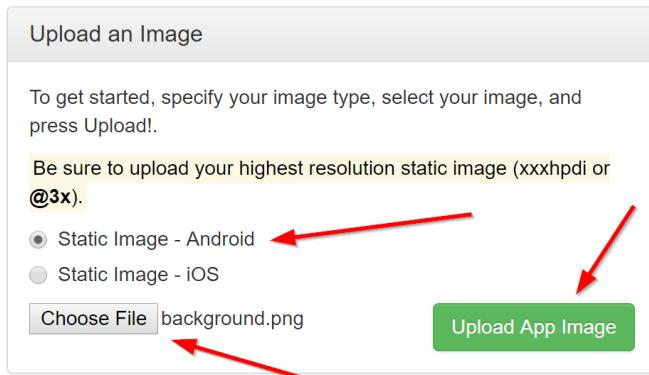


Figure 12.14 Our NativeScript Image Builder showing how to generate scaled background.png files.

After uploading the *background.png* file, you will receive back a zip file with *background.png* files in the appropriate *drawable-** folders (just like when we uploaded the app icon). Copy these files into their respective *drawable-** folders in the Pet Scrapbook.

REPLACING THE LOGO.PNG TO UPDATE THE LAUNCH SCREEN

Next, do the same for the logo.png file:

- Download the Pet Scrapbook app icon from <https://github.com/mikebranstein/TheNativeScriptBook/blob/master/PetScrapBook.png>.
- Rename the file to logo.png.
- Create scaled version of the image by uploading it to our image builder website.
- Copy the scaled logo.png files to their respective *drawable-** folders in the Pet Scrapbook app.

Let's look at the results! When the Pet Scrapbook launches, you'll see the updated launch screen for a split second before the main page appears (figure 12.15).

TIP Don't forget to rebuild your app with `tns build android` for the updated launch screen changes to appear.



Figure 12.15 The updated launch screen of the Pet Scrapbook app.

Now that we've updated the app icons and the launch screen of the Pet Scrapbook it is time to continue moving our way through the *AndroidManifest.xml* file. Our next stop is to look at how we can target different devices with our app.

12.3.2 Targeting various screen sizes

When you hear the words, Android app, what do you think of? A phone app? A tablet app? Both?

NOTE The Android app ecosystem goes way beyond phone and tablet apps: there are smart watches and even Android TV.

Wow! You probably weren't considering writing apps for watches or your TV before now. But even if you were, there's a configuration setting that tells your Android app which screen sizes you want to support.

We'd like the Pet Scrapbook to run on as many devices as possible, which means that it needs to support multiple screen sizes. Let's revisit the *AndroidManifest.xml* file and check out the *supports-screens* element (listing 12.3).

Listing 12.3 The *AndroidManifest.xml* file showing the default screen sizes that the Pet Scrapbook supports

```

<supports-screens
    android:smallScreens="true" // #A
    android:normalScreens="true" // #B
    android:largeScreens="true" // #C
    android:xlargeScreens="true"/> // #D
#A Support screens about 2 – 3 inches
#B Support screens about 2 – 5 inches
#C Support screens about 4 – 7 inches
#D Support screens about 7+ inches

```

The `supports-screens` element in the `AndroidManifest.xml` file is used by the Android runtime to specify the screen size compatibility for apps. Once again, you'll find that the NativeScript CLI made your job easy because it created default screen sizes for you: small (as described by `android:smallScreens`), normal (`android:normalScreens`), large (`android:largeScreens`), and extra-large (`android:xlargeScreens`).

NOTE Thanks, NativeScript CLI. You made my job easy...I think. It might be easy to turn support on and off for a specific screen size, but what does small, normal, large, and extra-large actually mean?

Hold on. Let's not diss the CLI, because it's really an Android issue. Unfortunately, the screen size properties used by Android are not descriptive. But, don't worry: table 12.2 breaks down the Android screen sizes and how they relate to various screen resolution DPIs.

Table 12.2 Android screen size properties and corresponding screen resolution DPIs

Screen Size	ldpi (120)	mdpi (160)	hdpi (240)	xhdpi (320)
Small	QVGA (240x230)		QVGA (480x640)	
Normal	WQVGA (240x400)	HVGA (320x480)	WVGA (480x800), (600x1024)	(640x960)
Large	WVGA (480x800)		WVGA (480x800), (600x1024)	
Extra-large	WSVGA (1024x600)	WXGA (1280x800), XGA (1024x768), WXGA (1280x768)	(1536x1152), (1920x1152), (1920x1200)	(2048x1536), (2560x1536), (2560x1600)

TIP If you would like to read about the `supports-screen` element in detail, you can review the official Android documentation at <https://developer.android.com/guide/topics/manifest/supports-screens-element.html>.

Ok. Table 12.2 is still a bit overwhelming, and truthfully, we'll never remember these details. The good news is you don't need to remember them either. Luckily, NativeScript is our saving grace because it handles the layout of each page of the Pet Scrapbook app for us; therefore, by default NativeScript has

set all the screen size properties to true so our app should adjust appropriately on all different devices and resolutions!

We could set one of the supported sizes to false (if we wanted). If we did set one to `false`, our app would follow the Android screen compatibility guidelines found at <https://developer.android.com/guide/practices/screen-compat-mode.html>.

As mentioned before, however, it's not needed or recommended to set any of these to false because NativeScript handles the layout and sizing of page elements intelligently for us.

12.3.3 Versioning

As you've been learning NativeScript and building the Pet Scrapbook app, we've continually added features, refactored app code, and changed the app significantly. When we release the Pet Scrapbook app to the Google Play store, we'll continue to add new features. What we're trying to say is that an app is never finished: there's always another version. But, you may be thinking, "How do I track the version of an app?"

App versioning can be a complex topic, but we've distilled it down to the basics for you. The Google Play store has two specific properties it looks at in the *AndroidManifest.xml* file. Listing 12.4 shows the manifest element that is used to control an app's version.

Listing 12.4 The AndroidManifest.xml file showing the versioning properties in the manifest element

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android" // #A
    package="__PACKAGE__"
        android:versionCode="1" // #B
        android:versionName="1.0"> // #C
#A The manifest node is the top-level element of the AndroidManifest.xml file
#B An internal version number that users never see
#C An external version string that users see in the Google Play store
```

An Android app's version is composed of two numbers: an internal version code (`android:versionCode` attribute) and an external version name (`android:versionName` attribute).

DEFINITION The version code is an integer (formatted as a string) and used as an internal app version number.

But, what does internal mean? Because version codes are internal, they are never shown to users and not reflected in the store. This means you can put any integer value in the version code attribute.

Because this will be the first release of the Pet Scrapbook app, we will leave this version number at "1." The version code can be treated like a build number, so it is best practice that every time we upload the Pet Scrapbook to the Google Play store we increment the number.

NOTE The Android system does not enforce restrictions on the version code attribute when uploading your app to the Google Play store. But, it is recommended that you increment it on each release of your app. Incrementing the version code with each release can help you stay organized.

The second property that controls an app's version is the externally visible version name.

DEFINITION The version name is a string that is shown to users when they are looking at an app in the Google Play store.

Android doesn't place any restrictions on the format of the version name, but most developers use a common technique of versioning called *semantic versioning*.

DEFINITION Semantic versioning is a popular, and generally-accepted way of publicly-versioning shared code libraries and packages using a MAJOR.MINOR.PATCH numbering scheme. To learn more about semantic versioning, visit <http://semver.org>.

For the Pet Scrapbook, we'll leave the default version name of 1.0 in place. We know it doesn't conform to the semantic versioning specification, but it's easy to leave it alone. Rest assured, when we publish a second version to the app store, we'll adjust the version name accordingly to respect semantic versioning.

12.3.4 App naming

The last AndroidManifest.xml file configuration option we'll be covering is the app's name. You may be thinking that an app's name is just cosmetic, but that doesn't mean it's not important. Imagine if we were to publish the Pet Scrapbook to the Google Play store and left its name to the default: *PetScrapbook* (figure 12.16).



Figure 12.16 The Pet Scrapbook app installed on an Android device, showing a missing space is missing in the app's name.

So, the default app name is *PetScrapbook*, because that's what we named the app when we created it using the `tns create PetScrapbook` CLI command. Normal users may not notice the missing space in the app's name, but any right-minded developer would immediately notice the faux-pas. Bottom line: don't forget to update your app's name.

Let's fix this issue. App names are stored in another file named *strings.xml*, which is in the *App_Resources/Android/values* folder.

NOTE The `strings.xml` file isn't created by the NativeScript CLI by default. Create a file named `strings.xml` and place it in the `App_Resources/Android/values` folder.

Listing 12.5 shows the contents of the `strings.xml` file. Add this to your `strings.xml` file.

Listing 12.5 The strings.xml file showing the updated app name

```
<resources>
    <string name="app_name">Pet Scrapbook</string> // #A
    <string name="title_activity_kimera">Pet Scrapbook</string> // #B
</resources>
#A The string resource for the Pet Scrapbook app name
#B The string resource for the activity app name
```

There's not much going on in the `strings.xml` file, and pragmatically, you don't need to know the details of how this file works. We're not going to explain details here, so just ensure you copy the XML into the `strings.xml` file.

TYING TOGETHER STRINGS.XML AND ANDROIDMANIFEST.XML

One item to note is how the `AndroidManifest.xml` file references the `strings.xml` file located in the `values` folder. Let's start by looking at the application element of the manifest file again (listing 12.6).

Listing 12.6 The updated App_Resources/AndroidManifest.xml file to reference the new app name strings

```
<application
    android:name="com.tns.NativeScriptApplication"
    android:allowBackup="true"
    android:icon="@drawable/icon"
    android:label="@string/app_name" // #A
    android:theme="@style/AppTheme">
    <activity
        android:name="com.tns.NativeScriptActivity"
        android:label="@string/title_activity_kimera" // #B
        android:configChanges="keyboardHidden|orientation|screenSize">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <activity android:name="com.tns.ErrorReportActivity"/>
</application>
#A References the app_name element of string.xml
#B References the title_activity_kimera element of strings.xml, giving the app's main activity a name
```

Like the `@drawable` naming convention we've seen throughout this chapter, the `@string` naming convention tells Android to look for a file named `strings.xml`, and to look for a string named `app_name` and `title_activity_kimera`. Together, these two string values change the name of our app from PetScrapbook to Pet Scrapbook. Figure 12.17 shows the results.

NOTE Don't forget to rebuild app with `tns build android` to see the change.

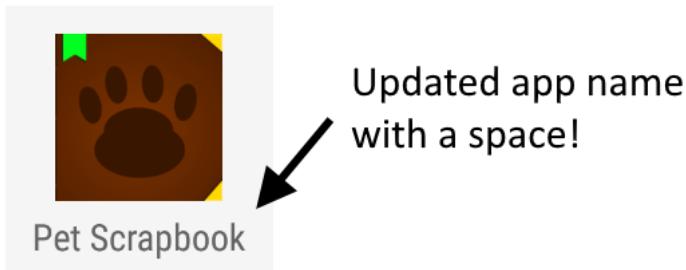


Figure 12.17 The Pet Scrapbook app installed on an Android device, showing an updated app name (with a space).

At this point, we've taken care of all the missing pieces and put a lot of polish on the Pet Scrapbook app. We're ready for deployment to the Google Play store! Let's look at the last steps we need to get our app into the store.

12.4 Building your app

The process of deploying a NativeScript app to the Google Play store is the same as deploying a native Android app to the store—signing and building a release version of the app and then uploading the app to the store.

NOTE There are many tutorials online that can walk you through uploading your app to the store and navigating through the app submission process, so we're not going to cover this part. If you're looking for specific guidance on the app submission process, start by checking out Google's official documentation at <https://developer.android.com/distribute/google-play/start.html>.

Even though we aren't going to walk you through the store submission process, we think it's critical that you understand how to build a release version of your app and then sign it.

DEFINITION A release version is a compiled version of an app, optimized for submission to the app store, and for running on physical devices. We're not going to go into the details of release version, but you can learn more by reading the official Android documentation at <https://developer.android.com/studio/publish/preparing.html>.

Before we get started with building a release version of the Pet Scrapbook, we'll need to cover some basics of app security on Android.

12.4.1 Digital signatures

As you develop and test your app on Android emulators, the emulator imposes a relatively low level of security on you and your app. But, when you enter the realm of physical devices and the Google Play store, the security gets cranked up. In short, every app built in release mode and deployed to a physical device must be capable of being absolutely (and without a doubt) verifiable with a digital signature.

DEFINITION A digital signature is a mathematically-proven mechanism for showing the authenticity of digital documents, codes, or binaries. When a digital signature is applied to something, it is often referred to as having been digitally signed. If you would like to read further information about signing your app for Android, you can review the official information at <https://developer.android.com/studio/publish/app-signing.html>.

All Android apps installed on a physical device are digitally signed, meaning that users of the app (and Google) have reason to believe the app is authorized for installation by Google and the app creator. Furthermore, because apps are digitally signed, users can be assured the app has not been tampered with on the device.

Now that you know just enough about app security to be dangerous, let's ramp it up another level. To digitally sign an app, you'll need a set of keys, and these keys are stored in a special file called a *keystore*.

12.4.2 Generating a keystore file

To create the keystore, we'll use a command line tool called *keytool* that comes with the Android SDK. If you've been following along with us throughout the book, you'll already have keytool installed by default.

Let's start by running the following command from terminal or command prompt:

```
keytool -genkey -v -keystore petscrapbook.keystore -keyalg RSA -keysize  
2048  
-validity 10000 -alias petscrapbook
```

NOTE If you're interested in the details of the keytool command line tool, you can view the complete documentation by running *keytool -help* or by reading more at <http://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html>.

When you run the *keytool* command, you'll be prompted to fill out some information such as your name, organization, and address, as shown in figure 12.18. Even though you don't have to fill out *all* the information it's recommended to fill out *at least* the name and organization. By supplying a name and organization, it helps others identify you as a valid creator of the Pet Scrapbook app.

TIP When running the *keytool* command, the keystore file will get created in the folder that you are running it from. It is recommended to store the file in a safe place after creation. If you are using source control that is public facing for your Pet Scrapbook project, be sure not to check in the keystore file. If someone were to gain access to your keystore file, they would be able to potentially use it to sign their applications.

```
Nicks-MBP-2:PetScrapbook Nick$ keytool -genkey -v -keystore petscrapbook.keystore -keyalg RSA -keysize 2048 -validity 10000 -alias petscrapbook
[Enter keystore password:
Re-enter new password:
What is your first and last name?
[Unknown]: Brosteins
What is the name of your organizational unit?
[Unknown]: Brosteins
What is the name of your organization?
[Unknown]: Brosteins
What is the name of your City or Locality?
[Unknown]:
What is the name of your State or Province?
[Unknown]:
What is the two-letter country code for this unit?
[Unknown]:
Is CN=Brosteins, OU=Brosteins, O=Brosteins, L=Unknown, ST=Unknown, C=Unknown correct?
[no]: y

Generating 2,048 bit RSA key pair and self-signed certificate (SHA256withRSA) with a validity of 10,000 days
    for: CN=Brosteins, OU=Brosteins, O=Brosteins, L=Unknown, ST=Unknown, C=Unknown
Enter key password for <petscrapbook>
[RETURN if same as keystore password]:
[Storing petscrapbook.keystore]
```

Figure 12.18 The additional information requested by the keytool to create the Pet Scrapbook keystore file used to sign the release version of the app.

HOW MANY KEYSTORE FILES DO YOU NEED?

There's no official rule from Google stating that you can't use the same keystore file to digitally sign all of your apps. But, we don't recommend using a keystore file more than once.

TIP Create a new keystore file for each of your apps.

We recommend creating a new keystore file for each app you plan to publish to the Google Play store. Yes, it's more files to keep track of and secure, but it could pay off if one of your keystore files is compromised. Imagine that a hacker breaks into your computer and steals your one and only keystore file. With that file, they'd be able to impersonate you, updating any of your apps. The good news is that there's a way to essentially *disable* the stolen keystore file, but it also requires you to update all the apps digitally signed with the keystore. If a separate keystore file is used for each of your apps, you still have to do some cleanup, but that cleanup isn't nearly as extensive.

12.4.3 Creating a release build of the Pet Scrapbook

Now that we have created the keystore file, we need to create the release build using the NativeScript CLI. This is done by running the `tns build android` command with several command line parameters, as follows:

```
tns build android --release --key-store-path petscrapbook.keystore
--key-store-password myPassword --key-store-alias petscrapbook
--key-store-alias-password myPassword
```

Running the `tns build android` command with the `--release` parameter will build the Pet Scrapbook release build. When the `--release` parameter is used, the key-store parameters must also be specified. There are several key-store related parameters, but they all relate to the keystore file we just created.

NOTE When you specify key-store parameters to the `tns build android` command, be sure to use the same passwords you used when creating the keystore file.

After running the build command, the NativeScript CLI will create an APK file, which is the file that is submitted to the Google Play store. By default, the file will be output to the `platforms/android/build/outputs/apk` folder. Figure 12.19 shows the .apk file.

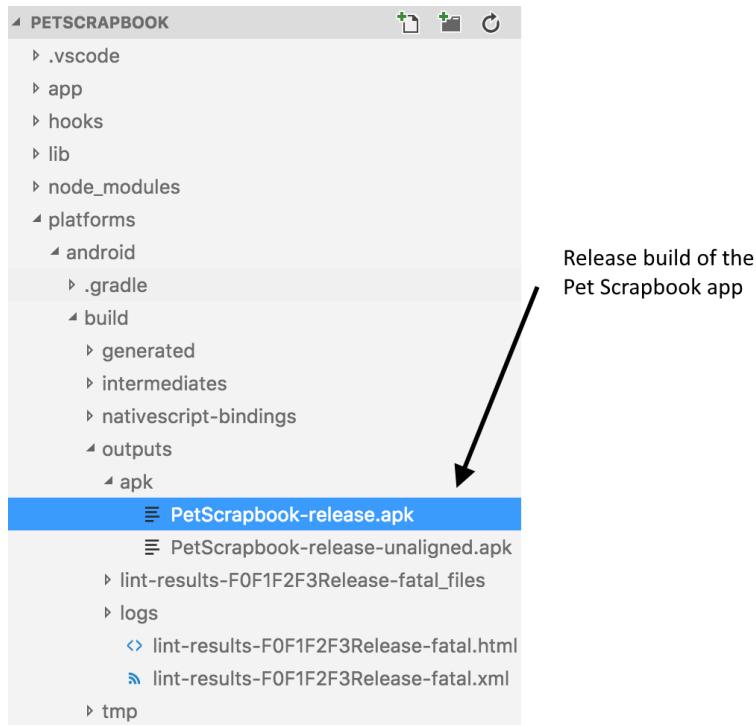


Figure 12.19 The location of the Pet Scrapbook APK file after building the release package of the app.

The Pet Scrapbook app is now complete and ready to be uploaded to the Google Play store! We don't want to stop you here, so if you're on a roll, you're welcome to submit your own version of the Pet Scrapbook to the Google Play store. To get started, visit <https://developer.android.com/distribute/googleplay/start.html>. And, if you do publish an app, be sure to let us know!

12.5 Summary

In this chapter, you learned to do the following:

- Create an app icon for Android.
- Create a launch screen for Android.
- Target an app for different Android screen sizes.
- Version and name your app on Android.

- Create a release APK that can be uploaded to the Google Play store.

12.6 Exercise

1. Change the name of the Pet Scrapbook icon from icon.png to petScrapbookIcon.png
2. Disable small screen support

12.7 Solutions

1. Update all the files in the drawable folders (drawable-ldpi, drawable-hdpi, etc.) from icon.png to petScrapbookIcon.png
 - a. Update the AndroidManifest.xml file
 - i. Change `android:icon="@drawable/icon"` to
`android:icon="@drawable/petScrapbookIcon"`
 - b. Rebuild the solution with `tns build android`
2. Update the supports-screens section in the AndroidManifest.xml to

```
<supports-screens  
    android:smallScreens="false"  
    android:normalScreens="true"  
    android:largeScreens="true"  
    android:xlargeScreens="true"/>
```

13

Preparing an iOS app for distribution

This chapter covers:

- Finalizing your app with a custom icon, launch images, and name
- Using the NativeScript CLI to create an Xcode project

In the last chapter, you learned how to get your app ready for publishing to the Google Play store. Because NativeScript is a platform-agnostic framework, much of what you learned in the last chapter applies to getting iOS apps ready for distribution through Apple's App Store. In fact, the core concepts are the same: you still need to work in the `App_Resources` folder to customize app icons, create a splash screen, and rename and version your app. But, you just do it differently for iOS.

Over the next two chapters, you'll learn how to prepare your app for publishing on Apple's App Store.

NOTE Whoa! Two chapters for iOS deployment? Yep. But, please don't panic. There is a lot to learn about iOS app security and deployment. When we started developing mobile apps, we felt lost in iOS, and these two chapters are what we wish we had.

In this chapter, we'll focus on using features of the NativeScript platform and CLI to get your app ready. We'll follow-up in chapter 14 with a look at the core elements of iOS app security that you'll need to understand before publishing to the App Store. We'll also cover the fundamentals of `Xcode` that every NativeScript app developer should know.

DEFINITION `Xcode` is Apple's IDE (or integrated development environment) used to create, build, and submit iOS apps to the App Store.

Before we jump in, we want to address the elephant in the room. To build and publish iOS apps, you need a Mac. And, there's no way around it.

TIP Just because you need a Mac to build iOS apps, it doesn't mean you need to drop \$2000 and buy a fancy new laptop. Services like MacinCloud (<http://www.macincloud.com>) and vmOSX (<https://virtualmacosx.com>) allow you to rent a virtual Mac, including pay-as-you-go plans. You can also pick up a refurbished Mac mini for under \$500: check out Apple's certified refurbished website because the deals change daily!

Now that we've covered how to get your hands on a Mac, let's get the Pet Scrapbook ready for the App Store!

13.1 Transforming your app code into an iOS app

In chapter 12, you learned about a handful of things that should be done before you publish your app to the Android store: changing the name and versioning your app, adding app icons, adding a splash screen, and supporting multiple device sizes. For Android, these changes were managed through several files under the *App_Resources* folder. Preparing your app for the iOS platform is the same: the files and folders within the *App_Resources* folder are key.

Before we take a closer look at the *App_Resources* folder, we think it's important to revisit the build process at a high-level and answer the question: how does a NativeScript app (which is just a collection of XML, JavaScript, and CSS) turn into an iOS app?

Back in chapter 1, we briefly discussed that the NativeScript CLI is what bridges the gap between your app code and an iOS app (figure 13.1).

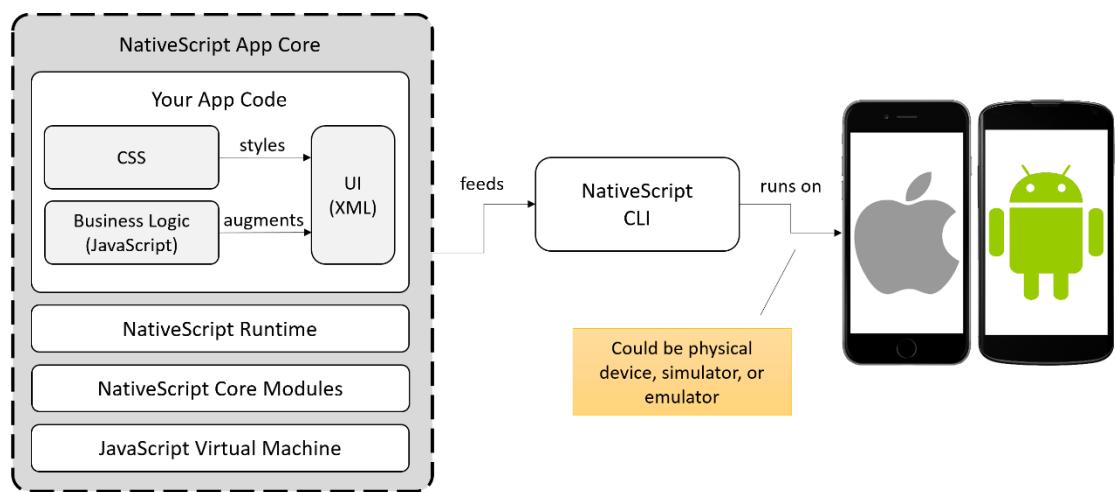


Figure 13.1 The NativeScript CLI is responsible for transforming your app code (XML, JavaScript, and CSS) into an iOS app.

At a high level, the NativeScript CLI is responsible for transforming your app code (XML, JavaScript, and CSS) into an iOS app. But, there's a lot more going on behind the scenes. Don't worry, you don't need to know all the details, but it can be helpful for you to learn the basics.

To transform your app code into an iOS app (.ipa file), the CLI performs several actions in two phases: the prepare phase and the build phase (table 13.1).

Table 13.1 The two execution phases for transforming NativeScript app code into an iOS app

Phase	Description
Prepare	A multi-step process that copies app files to the iOS platforms folder, merges app images into Xcode-specific resources, creates an Xcode project file, and merges configuration files into the Xcode project
Build	Builds the Xcode project created in the prepare phase, producing an iOS app (.ipa file)

13.1.1 The prepare phase

The *prepare* phase is the first phase of the transformation process, and it's responsible for creating an Xcode project that contains your app files.

TIP The Xcode project produced by the prepare phase is stored in the platforms/ios folder of your NativeScript app.

The prepare process is lengthy and detailed, so we've summarized the process into five steps:

- Step 1: Copy app files. The app files you've created (XML, JavaScript, and CSS) are copied to the Xcode project location.
- Step 2: Copy App_Resources files. The files in the App_Resource folder (app icons, images, launch screens, configuration files, and so on) are copied to the Xcode project location.
- Step 3: Merge images and configure launch screen. The copied images and launch screen are merged and formatted into a format and structure that's understood by the Xcode project.
- Step 4: Create Xcode project. An Xcode project file is created that references the merged images and launch screen.
- Step 5: Merge configuration files. Configuration files copied from the App_Resources folder are merged into the Xcode project.

13.1.2 The build phase

After an Xcode project is prepared, the *build* phase is responsible for taking the Xcode project file and producing the iOS app (.ipa file). There's not much to talk about in this phase, because the CLI simply invokes a command-line executable that builds the Xcode project.

NOTE There's a certain beauty to the NativeScript CLI. It doesn't do anything special once an Xcode project is created. In fact, at that point, everything is native. Xcode builds your app.

At this point, you may be thinking, "That's it?" Yeah. It's rather anti-climactic. The prepare and build phases are straightforward and rely on the underlying toolsets of Xcode to do the really hard stuff: producing the native iOS app (.ipa file).

Before we take a deeper dive into the *App_Resources* folder, let's tie this all together by looking at how the prepare and build phases are invoked.

13.1.3 Using the CLI to prepare and build your app

Throughout the book, we've covered three CLI commands that tie all of this together (`tns platform add ios`, `tns prepare ios`, and `tns build ios`). But it's valuable to look at them again (briefly) now that you understand the prepare and build phases.

NOTE Much of what we're covering here applies to both iOS and Android platforms, so you'll be able to apply what you learn here if and when you build an Android app.

- `tns platform add ios`: this command starts the entire process by creating the *platforms/ios* folder in your app, and populating the folder with the iOS SDK files that we'll need to later build the app.
- `tns prepare ios`: this command executes the prepare phase of the app transformation process.
- `tns build ios`: this command executes the build phase of the app transformation process.

It's no coincidence there are prepare and build commands as part of the CLI. That's because these commands directly equate to the two phases of the transformation process: `tns prepare ios` executes the prepare phase, and `tns build ios` executes the build phase.

TIP Don't feel frustrated thinking that you need to type in three separate commands to build your app, because you don't. Each of the commands will run the previous command automatically. For example, running `tns build ios` will add the *platform/ios* folder, prepare the Xcode project, then build your app.

Ok. Now that you've learned how the CLI ties everything together, let's dive deeper into the *App_Resources* folder and find out how you can customize your app.

13.2 Finalizing your app

In the previous section, you learned how the prepare phase (and the `tns prepare ios` command) works to create an Xcode project and merge custom settings and configurations from files in the *App_Resources* folder.

When you create your app with the NativeScript CLI, the *App_Resources/iOS* folder is populated with iOS-specific files. When modified, these files can affect the behavior of your app, allowing you to control

the app name, iOS icon, launch image, and support device sizes. You can use the default iOS App_Resources files without any modification and your app can be published to the App Store. But, the default icon (right) and launch screen (left) probably won't be sufficient (image 13.2).



Figure 13.2 The default NativeScript launch screen (left) and app icon (right).

Don't get us wrong – the NativeScript app icon and launch screen look good, but mobile apps are about creating a great experience for users. So, you'll want to customize these items before publishing your app.

TIP Don't publish your app to the App Store without customizing the app name, icons, and launch screen.

13.2.1 *Naming your app*

The first customization you should make is to change your app's name. If you're anything like us, when we create our apps using the CLI, we like to keep the app name simple, like PetScrapbook: `tns create PetScrapbook`. You'll recall this creates a folder named PetScrapbook, and it also names your app PetScrapbook (figure 13.3).

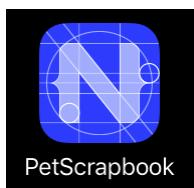


Figure 13.3 The name displayed when installed is PetScrapbook, not Pet Scrapbook.

Notice the app name is PetScrapbook, without a space in the name. We're a bit pedantic: the app name should have a space in it and read Pet Scrapbook. Let's fix the name by updating the *Info.plist* file, specifically the *CFBundleDisplayName* value.

DEFINITION The *Info.plist* stands for information property list, and is a configuration file used by Xcode. The file is a series of XML-formatted key-value pairs. The file can be edited via the Xcode UI, or with a text editor (because it's just XML). All iOS apps have an *Info.plist* file. For more information on the *Info.plist* file, check out Apple's official documentation at <https://developer.apple.com/library/content/documentation/General/Reference/InfoPlistKeyReference/Articles/AboutInformationPropertyListFiles.html>.

Locate the *CFBundleDisplayName* key and value in the *Info.plist* file, as follow:

```
<key>CFBundleDisplayName</key>
<string>${PRODUCT_NAME}</string>
```

The default value is *\${PRODUCT NAME}*, an iOS build-time dynamic variable. You don't need to know about dynamic variables, so we're not going to explain them. Instead, replace *\${PRODUCT NAME}* with Pet Scrapbook:

```
<key>CFBundleDisplayName</key>
<string>Pet Scrapbook</string>
```

Re-build and install the app your app with `tns run ios`. If you exit your app to the home screen, you'll notice the app's name has changed. Figure 13.4 shows the app's name without a space on the left and with the space on the right. The change is subtle, but noticeable.



Figure 13.4 Changing the *CFBundleDisplayName* key in the *Info.plist* file changes the app name to Pet Scrapbook (notice the space). The app on the right has a subtle space in the name.

13.2.2 Versioning

An important step of publishing an app is establishing a *versioning scheme*.

DEFINITION A versioning scheme is an established system for keeping track of an application's version. In iOS, an app's version is tracked via two numbers: the app's version number and build number. Together, these two numbers uniquely describe the app version.

The versioning scheme, version number, and build number are important because every app submitted to the App store must have a unique version.

ESTABLISHING A VERSIONING SCHEME

Apple provides iOS developers with some general guidance around app versioning (see https://developer.apple.com/library/content/technotes/tn2420/_index.html for this guidance). Versioning with an app version number and an app build number can be rather confusing. Rather than describing all the intricacies of establishing a versioning scheme, we'll recommend a way to version your app. But, before we jump into our recommendations, we'll introduce you to the basics of the app's version number and build number.

DEFINITION The version number is a series of integers separated by dots, like 1.0.0 or 2.5.1. Each time an app is submitted to the iTunes store, it must be assigned a unique version number, and with each new release, its version number must increase. For example, if an app has been published with version 1.3.4, the next released version must be greater than 1.3.4. 1.3.5, 1.4.0, and 2.0.0 are version numbers greater than 1.3.4. Likewise, 1.0.0, 1.2.7, and 0.8.9 are version numbers less than 1.3.4.

Because version numbers always increase when a new version of an app is released, it's easy to tell when an app has been updated.

DEFINITION The build number of an app is intended to identify a build of an app, targeting a specific version. For example, for app version 1.0.0, we may build once, creating build 1, then identify a problem requiring us to build a second and third time. The second and third builds create build number 2 and 3. Once the app is released to the iTunes store, we begin working on version 2.0.0, which has its first, second, and third builds numbered 1, 2, and 3. Like version numbers, build numbers follow the same rules on numbering, but have one difference: a build number can be reused.

Now that you know the basics about version and build numbers, let's get down to our recommendations. Truth be told, Apple's guidance on app versioning is very permissive, allowing you to do crazy things (in our opinion) like having a version number of 7.3.1, with build numbers of 0.3.1, 7.3.1, and 9.0.0.0. Likewise, a later app version (8.0.3 for example) may also have the same build numbers.

Being permissive isn't bad, but it can be confusing if you're collaborating with others, and referring to build number 7.3.1, which could be used across several versions. It gets even more complicated because the build number can be identical to the app's version number!

There is a better way.

A REASONABLE VERSIONING SCHEME

You're welcome to version your apps in any way you like, but we prefer using a strict approach to help reduce confusion. We won't call this a best practice, but we think it's on its way to being considered a reasonable practice. Try it out, and if you don't like it, we won't shed any tears.

12. Set the version number of your app using semantic versioning, which uses MAJOR.MINOR.PATCH numbering scheme.
13. Start at version number 0.0.0.
14. Increment the PATCH number when you are publishing a bug fix.

15. Increment the MINOR number when you are adding a new feature in a backwards-compatible way for the end-user. Incrementing the MINOR number resets the PATCH number to 0.
16. Increment the MAJOR number when you are making a breaking change for the end-user. Incrementing the MAJOR number resets the MINOR and PATCH numbers to 0.
17. Align your build number scheme to the version number scheme, but with an additional BUILD number to the end: MAJOR.MINOR.PATCH.BUILD. For example, builds targeting version 1.0.0 will be numbered 1.0.0.1, 1.0.0.2, and so on until version 1.0.0 is published. Once published, the BUILD number resets to 0 with a new version.

DEFINITION Semantic versioning is a popular and generally accepted way of publicly versioning shared code libraries and packages using a MAJOR.MINOR.PATCH numbering scheme. To learn more about semantic versioning, visit <http://semver.org>.

VERSIONING NATIVESCRIPT APPS

Now that you've learned about app versioning on iOS, let's see how we can change the app version by editing the *Info.plist* file. To set the version and build numbers, we'll be changing two values: *CFBundleVersion*, which corresponds to the version number, and *CFBundleShortVersionString*, which corresponds to the build number.

Although it's a bit trivial at this point, let's set the version and build number of the Pet Scrapbook to 1.0.0. Find the *CFBundleVersion* value and change it to 1.0.0, as follows:

```
<key>CFBundleVersion</key>
<string>1.0.0</string>
```

Then set the *CFBundleShortVersionString* value to 1.0.0:

```
<key>CFBundleShortVersionString</key>
<string>1.0.0</string>
```

That's it, and we know: It's quite anti-climactic. And, unfortunately, there's not a great way out-of-the-box way to visualize the app version. But don't worry, we'll see how setting the app version in the *Info.plist* file flows to Xcode later in this chapter. Stay tuned.

TIP Even though NativeScript doesn't provide a built-in mechanism for getting an app's version number, the *nativescript-appversion* plugin does. Check it out in the npm package at <https://www.npmjs.com/package/nativescript-appversion>.

13.2.3 Adding icons

In chapter 7, you learned how to add images to your app, and specifically how to address images on a variety of devices with different DPIs. You might be wondering how this relates to an app's icon.

TIP App icons are images, and you use the same tools and strategies you used when creating images to create an app icon.

Because app icons are just images, you already know everything you'll need to resize and format the icons. But, you're still missing some of the details. Let's explore app icons in the *App_Resources* folder.

ICON LOCATION

App icons are stored in the `App_Resources/iOS/Assets.xcassets/AppIcon.appiconset` folder, and are named `icon-{size}.png`. Wow! There's a lot going on in that folder structure, so we'll break it down for you, starting with the `Assets.xcassets` folder, which holds various asset catalogs.

DEFINITION An asset catalog is a way of organizing app images, icons images, or launch screen images so they can be used across a variety of screen dimensions and DPIs.

Creating asset catalogs is straightforward, but you don't need to know the details because the CLI provisions an asset catalog for your icons and launch screen automatically. Let's get back to the `Assets.xcassets` folder. Now that you know about asset catalogs in a general sense, the name of the folder should make more sense: it contains assets for Xcode (often abbreviated as `xc`).

One of the folders within `Assets.xcassets` is the `AppIcon.appiconset` folder. This folder contains the definition of an asset catalog for app icons (figure 13.5).

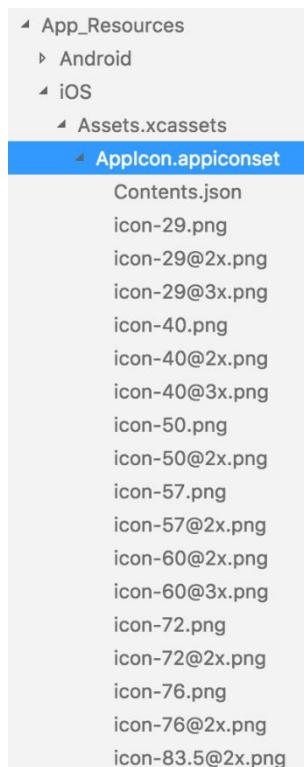


Figure 13.5 The AppIcon.appiconset folder, defining the asset catalog for app icons.

There are two items of interest within the `AppIcon.appiconset` folder: the `Contents.json` file and 17 images. The `Contents.json` file defines the various icon images, their purpose, and size in pixels. We're not going to delve into the exact contents of the file, instead what you need to know is that by replacing the images in this folder, you can affect an app's icon.

NOTE Yow! There are 17 images you need to replace to change the app's icon? Yep. But don't be worried, there are tools out there to help you to resize an icon into the various sized images (like the NativeScript Image Builder at <http://nsimage.brosteins.com>).

Now that you know about app icons, let's get to work and change the icon for the Pet Scrapbook. We'll be using our NativeScript Image Builder tool at <http://nsimage.brosteins.com> because we've discussed it in detail back in chapter 7, and it makes resizing app icons simple. We'll begin by downloading the Pet Scrapbook icon from our Github repository: <https://github.com/mikebranstein/TheNativeScriptBook/blob/master/Chapter13/PetScrapBook.png> (figure 13.6).



Figure 13.6 The Pet Scrapbook app icon.

TIP Make sure the app icon you create is the right shape and size. App icons must be square images, and 1024 x 1024 pixels.

CHANGING THE ICON

Using the NativeScript Image Builder tool, upload the `PetScrapBook.png` app icon, as shown in figure 13.7.

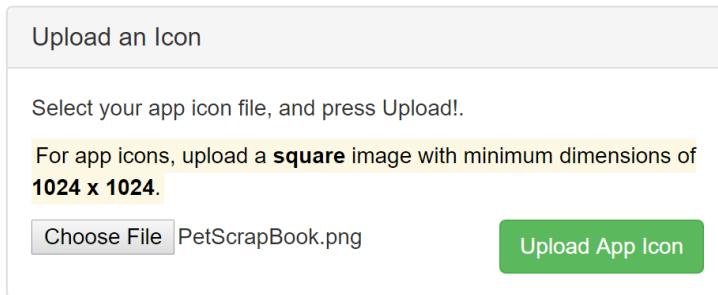


Figure 13.7 Uploading the Pet Scrapbook app icon to our image builder tool.

After a few seconds, the image builder tool will download a zip file containing the resized app icons. Open the zip file and navigate into the iOS folder, where you'll find 24 app icons (figure 13.8).

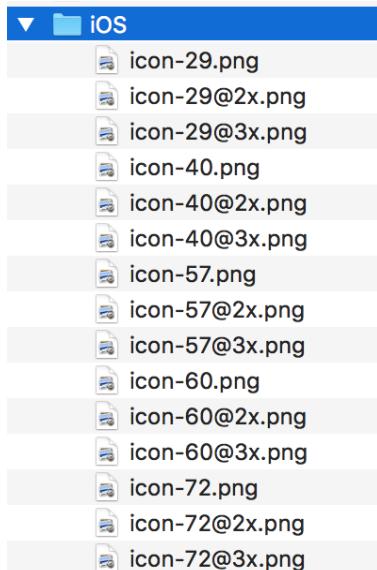


Figure 13.8 The iOS folder in the image builder zip file contains 24 app icons. Note the image is clipped to save room.

Copy the files with a name starting with *icon* into the *AppIcon.appiconset* folder, overwriting the existing files.

NOTE You may have noticed that the image builder tool creates more than 17 images: it creates 24 images. Two of the images (*iTunesArtwork.png* and *iTuneArtwork@2x.png*) can be used when

submitting your app to the iTunes store. This still leaves 5 more icons. You don't need these images technically, but the image builder tool creates 3 images for each icon size. It doesn't hurt to copy the extras into your project, because they'll be ignored.

Let's check out the new app icon (figure 13.9). Run `tns platform remove ios`, then `tns run ios` to use the new app icon.

NOTE Remember, when you remove a platform using the `tns platform remove` command, it deletes only the platform-specific app files from the *platforms* folder, not the customizations you made to the *App_Resources* folder (like the new app icon).



Figure 13.9 The Pet Scrapbook app after replacing the default app icons with the images created with our image builder.

WARNING Xcode caches app icon asset catalogs, so you need to remove the Xcode project created by NativeScript by running `tns platform remove ios`. Once removed, running `tns run ios` will rebuild the Xcode project with the updated app icon.

TIP It's always good to f you're still feeling a bit new to

13.2.4 Launch screens

Now that you know about asset catalogs and how to change the app icon, we'll use a similar approach to create launch screens.

DEFINITION Launch screens allow you to display a splash screen while your app loads.

Launch screens are a great way to brand your app and create a better user experience because it provides feedback to users, letting them know your app is loading. Even though launch screens can look like anything, it's standard to show your app's icon centered on the screen.

TIP Don't spend a lot of time and effort creating a launch screen. Launch screens should reflect your brand. Use your app's icon, centered on a solid background. Simple is better.

LAUNCH SCREENS VERSUS LAUNCH IMAGES

You may have heard about something called a launch image. Launch images and screens accomplish the same thing: displaying an initial splash screen as an app loads, but they do this in two different ways.

DEFINITION Launch images is a legacy method used in iOS 7 to display a splash screen by creating custom-sized images for the various screen sizes and device orientations.

DEFINITION Launch screens are the new method for displaying a splash screen. Supported in iOS 8, 9, & 10, this method allows you to create a single storyboard that will adjust to all screen sizes and device orientations automatically.

Because NativeScript supports iOS 7 and higher, it allows you to create both launch images and screens. But, we don't think this means you should spend the time to support both.

TIP If your app *absolutely must* target iOS 7, you'll have to create a launch image. Otherwise, skip iOS 7 and skip creating launch images. Focus on supporting iOS 8 and above, and create a single launch screen.

We feel strongly that you should not make launch images to support iOS 7, and here's why:

- It's tedious. How many device sizes and orientations are you going to support? Now create a custom image for each. No thanks.
- As of February, 2017, 79% of iOS devices run iOS 10, 16% iOS 9, and a whopping 5% run iOS 8 or earlier. That means iOS 7 is somewhere inside that 5%. See <https://developer.apple.com/support/app-store> for the stats.

With that said, we're not going to cover how to create launch images in this book. But, we don't want to abandon you if you're part of that 5% that needs to support iOS 7. If you need to support launch images, the official NativeScript documentation is a great resource. Check out <https://docs.nativescript.org/publishing/creating-launch-screens-ios#customizing-launch-images>.

EXPLORING LAUNCH SCREENS

The default NativeScript app template contains a pre-built launch screen storyboard and asset catalogs that store and organize images used on the storyboard.

DEFINITION A launch screen storyboard is an XML-formatted document describing the UI layout for the launch screen. In many ways, the storyboard is like a NativeScript page, except it contains iOS-specific XML markup to describe native iOS UI elements.

The launch screen storyboard is in the `App_Resources/iOS` folder, and is named `LaunchScreen.storyboard`.

TIP If you're curious, take a closer look at the launch screen storyboard. But you don't need to know anything of the specifics. NativeScript makes launch screens easy.

EXPLORING THE DEFAULT LAUNCH SCREEN STORYBOARD

The default launch screen storyboard included in the NativeScript app template creates a launch screen layout with the intent of doing the following two things:

1. To create a simple, lightweight UI with no moving elements that can be displayed quickly, per Apple's human interface guidelines at <https://developer.apple.com/ios/human-interface-guidelines/graphics/launch-screen>.
2. Display a solid background image stretched to fill the device's screen with a second image centered on top.

Now that you know a little more about the launch screen storyboard, let's look at how the two images (background and centered foreground) are organized inside of the *Assets.xcassets* folder

LAUNCH SCREEN BACKGROUND IMAGE

The *LaunchScreen.AspectFill.imageset* folder organizes images used as the background image of the storyboard (figure 13.10).

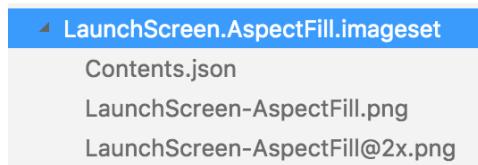


Figure 13.10 The *LaunchScreen.AspectFill.imageset* folder organized images used for the background image of launch screens.

You'll notice two launch screen image files and a *Contents.json* file. The images are just a solid color image. When displayed in various devices, this image will be stretched to fill the entire screen. The *Contents.json* file in this folder is like the one we looked at earlier for app icons: it describes the images contained in this folder. The internal workings of the file aren't important, so we won't cover it.

WARNING If you're customizing the launch screen background image, be aware that the image is stretched to fit a variety of screen sizes and dimensions. Because this is an image, you can make it more complex-looking with gradients, multiple colors, and so on. But we don't recommend it. Keep it a simple, solid color, and it will scale to every device.

LAUNCH SCREEN FOREGROUND IMAGE

The *LaunchScreen.Center.imageset* folder organizes images used as the centered, foreground image on the storyboard (figure 13.11).

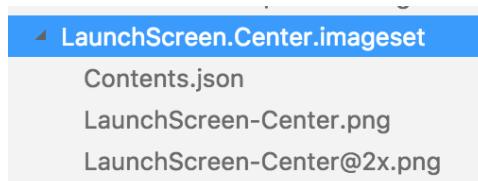


Figure 13.11 The *LaunchScreen.Center.imageset* folder organizes images used for the centered, foreground image of a launch screen.

The contents of this folder are almost identical to the background image's folder. There's nothing special going on in here, so all you really need to know is that whatever images you place in here will be used as the foreground image of an app's splash screen.

CUSTOMIZING THE LAUNCH SCREEN

Customizing the launch screen is straightforward: replace the images in the `LaunchScreen.AspectFille.imageset` and `LaunchScreen.Center.imageset` folders. Yeah, that's it. Let's update the Pet Scrapbook by changing the centered image to be the Pet Scrapbook app icon.

Looking in the `LaunchScreen.Center.imageset` folder, we need to provide two image files: an image file named `LaunchScreen-Center` and a 2x version of the same image. Again, we'll use the NativeScript Image Builder at <http://nsimage.brosteins.com> create the various images (figure 13.12).

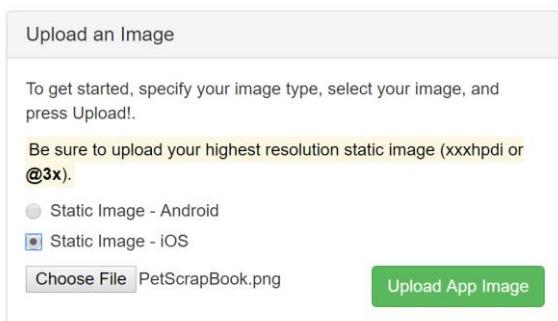


Figure 13.12 Reusing the Pet Scrapbook icon as the center image and uploading it to our image builder site.

Upload the Pet Scrapbook app icon we used earlier in this chapter to the image builder, using the iOS static image option. The image builder will return a zip file containing images re-sized to 1x, 2x, and 3x sizes. Rename the images to match the `LaunchScreen-Center` file naming convention and copy the 1x and 2x images to the `LaunchScreen.Center.imageset` folder.

With that change, remove the ios platform with `tns platform remove ios`, and run your app with `tns run ios`. You should now see the updated launch screen when the Pet Scrapbook starts (figure 13.13).

TIP The iOS simulator often caches the launch screen even after a rebuild and reinstallation. Restart the simulator to clear the cache.



Figure 13.13 The updated launch screen for the Pet Scrapbook.

ADVANCED LAUNCH SCREEN CUSTOMIZATION

If you're feeling limited by the launch screen storyboard included in the default NativeScript app template, you have option. The storyboard can be customized by changing the UI elements in the *LaunchScreen.storyboard* file. We're not going to cover this in this book, but we think it's important enough to briefly mention it. The sky's the limit here, but be sure to keep it simple and follow Apple's guidance at <https://developer.apple.com/ios/human-interface-guidelines/graphics/launch-screen>.

13.2.5 Supporting device orientations

The last point when preparing your app for the iTunes store is to establish the device orientations your app supports. You may recall from chapter 3 that it's possible to create pages that specifically target portrait or landscape mode by using a file-naming convention (remember `.port` and `.land`). Even though this is possible, it's a lot of work to create two different page layouts for every page of your app.

TIP To save yourself some time, decide whether you'll support portrait, landscape, or both view early in your app development process. And don't worry that it may seem you're cutting corners. Some of the most popular apps restrict you to an orientation (see Twitter and Facebook as examples).

CONFIGURING SUPPORTED ORIENTATIONS

To configure the device orientations your app supports, find the `UISupportedInterfaceOrientations` and `UISupportedInterfaceOrientations~ipad` keys in the `Info.plist` file. The values of these two keys are an XML-formatted array of possible screen orientations (listing 13.1).

Listing 13.1 Device orientation settings in the Info.plist file

```

<key>UISupportedInterfaceOrientations</key>           //##A
<array>
  <string>UIInterfaceOrientationPortrait</string>        //##A
  <string>UIInterfaceOrientationLandscapeLeft</string>  //##A
  <string>UIInterfaceOrientationLandscapeRight</string> //##A
</array>
<key>UISupportedInterfaceOrientations~ipad</key>       //##B
<array>
  <string>UIInterfaceOrientationPortrait</string>        //##B
  <string>UIInterfaceOrientationPortraitUpsideDown</string> //##B
  <string>UIInterfaceOrientationLandscapeLeft</string>    //##B
  <string>UIInterfaceOrientationLandscapeRight</string>   //##B
</array>
#A This key applies to iPhones
#B The ~ipad key applies to iPads

```

To add or remove support for a specific screen orientation on an iPhone, add or remove a value array string from the `UISupportedInterfaceOrientations` key. To affect supported device orientations on an iPad, add or remove items entries from the `UISupportedInterfaceOrientations~ipad` key.

If you'll recall from previous chapters, we created both phone and tablet views of the Pet Scrapbook. But, after some testing, the iPhone works best in portrait, and iPad in landscape. Let's make these changes by removing several of the options. Listing 13.2 shows the changes.

Listing 13.2 Pet Scrapbook Info.plist file supporting portrait on phones and landscape on tablets

```

<key>UISupportedInterfaceOrientations</key>
<array>
  <string>UIInterfaceOrientationPortrait</string>
</array>
<key>UISupportedInterfaceOrientations~ipad</key>
<array>
  <string>UIInterfaceOrientationLandscapeLeft</string>
  <string>UIInterfaceOrientationLandscapeRight</string>
</array>

```

TIP When supporting landscape view, we recommend including `UIInterfaceOrientationLandscapeLeft` and `UIInterfaceOrientationLandscapeRight`. It's not right or wrong to support one or the other, but we've met some people that are very particular in the way they hold their iPads when in landscape orientation. Left-handed people tend to hold their iPad so the Home button is on the left, nearest their left hand, and the opposite is true of righties.

That's it! You're finished and ready to get your app into the App Store. Moving to the App Store is a big step, especially if you're not familiar with the process (and it *is* a process). Don't worry though, we've been in your shoes before, and we'll guide you through everything you'll need to know in the next chapter.

13.3 Summary

In this chapter, you learned:

- How to use configuration files in the *App_Resources* folder to customize an app's icons and launch screen.
- How the NativeScript CLI uses a two-phase process to create an Xcode project.
- Launch screens (not launch images) are the preferred way of creating a launch screens for your iOS app because a single launch screen can be used across various devices.
- How to configure the screen orientations your app supports by modifying the *UISupportedInterfaceOrientations* and *UISupportedInterfaceOrientations~ipad* keys in the *Info.plist* file.

13.4 Exercises

Using what you learned in this chapter, do the following:

1. Create a new NativeScript app with an app id of *org.nativescript.{random name}*.
2. Create an app icon image and scale it to the various file sizes to support deployment to the App Store.

13.5 Solutions

1. Use the CLI to create the new project: `tns create sampleapp`. NativeScript will automatically set the app id to *org.nativescript.sampleapp*.
2. To create an app icon image for various device form factors:
 - a. Create a PNG image that is 1024 x 1024 pixels.
 - b. Using <http://nsimage.brosteins.com>, upload the image to the app icon area.
 - c. Open the .zip file produced by our site and inspect the files in the iOS folder: they should be named and sized appropriately for iPhone and iPad devices required by the App Store.

14

iOS security and building your app with Xcode

This chapter covers:

- How to use app IDs, certificates, registered devices, to create provisioning profiles for apps
- Using Xcode to build and archive NativeScript apps

In the last chapter, you learned how to get your app ready for the App Store by using the NativeScript framework and CLI. Because support for configuring an app's icons, launch screens, and support for multi-device screen orientation is built into NativeScript, you don't need to use the native iOS tools (like Xcode) to get your app ready. Even though we didn't need to use Xcode to get your app ready, we believe it's important to learn the fundamentals of iOS app security and Xcode.

Admittedly, a lot of tutorials online are focused on app security with Xcode, but none of them were made for the beginning iOS app developer who is focused on building apps with NativeScript. Until now, you've learned the NativeScript way of doing things, but getting your app into the App Store requires some knowledge and background in Apple's way of doing things.

So, if you're new to mobile app development on iOS, this chapter is for you. In this chapter, you'll learn what an Apple ID and developer account are, then learn the essentials of iOS app security: teams, app IDs, certificates, registered devices, and provisioning profiles. After you've built a solid foundation, we'll introduce you to Xcode, where you'll learn exactly what you need to know to take a NativeScript app from the CLI to App Store upload.

Let's get to it!

14.1 Building your app

In the last chapter, you learned how to configure and customize your NativeScript iOS app by modifying files within the `App_Resources` folder. Now, let's look at how the NativeScript CLI uses this information to transform your app into a native iOS app using Xcode.

DEFINITION Xcode is Apple’s IDE (or integrated development environment) used to create, build, archive, and submit iOS apps to the App Store.

NOTE Throughout this chapter we discuss the App Store, where you can purchase and download iOS apps for your iOS device. When we say *publishing to the app store*, we’re also referring to something called iTunes Connect, a website dedicated to managing your app store submissions. We’ll talk about both the app store and iTunes Connect later, but be aware of the difference and our shortened form of *App Store* that means both app store via iTunes Connect.

Until now, you haven’t needed to run Xcode to create and test your app in a simulator. In fact, if you don’t want to publish your app to the App Store or install it on a physical device, you needn’t go any further. We’ll see you in the next chapter!

NOTE You need to use Xcode if you want to install your app on a physical device or publish it to the App Store only. If you’re creating your app to run in your simulator just for fun, there’s no need to explore Xcode.

Ok. We’ll assume you’re still with us because you’re ready to build your app in Xcode, or you’re genuinely interested. Either way, we’re glad you’re still here!

NOTE Building, archiving, publishing? What’s the difference? In this chapter, we’ll discuss building, archiving, and publishing iOS apps. From afar, these three concepts are related, but happen in a distinct order. *Building* (or compiling) an app happens first, turning the raw code into a compiled binary. *Archiving* happens second, which bundles the compiled binaries into a package that can be installed on a physical device or published to the App Store. *Publishing* happens third, and is an optional upload (or submission) of an archived app to the App Store.

DO YOU REALLY NEED XCODE?

Before we jump into Xcode, let’s clear something up. You might be thinking, “Why do I need Xcode when the NativeScript CLI has gotten me this far? My app runs on a simulator already. Why can’t it run on a physical device?”

We applaud your logic and sensibility.

Actually, you don’t need to use the Xcode *GUI* to build an iOS app: the NativeScript CLI can build your app and publish it to the App Store. But, we think it’s easier to use Xcode, so we’re going to be walking you through that route.

NOTE We think it’s easier to use Xcode to build and publish your app because errors or warnings are more prevalent, and the GUI feels more user-friendly to us (once you learn your way around). We also think we’d be doing you a disservice if we didn’t teach you how to get your app ready for the App Store with the Xcode GUI.

If you’re interested in how to use the NativeScript CLI to publish your app, check out the official NativeScript documentation at <https://docs.nativescript.org/publishing/publishing-ios-apps>.

In the next several sections, we'll explore how to use Xcode to build and publish your apps, but before we can get there, we need to cover some basics of Apple's app security model. At first, you may think we're going too deep, but if you've never published an iOS app, this is essential knowledge. Stick with us.

14.1.1 Exploring iOS app security

As you develop and test your app on the iOS simulator, Apple imposes a relatively low level of security on you and your app. But, when you enter the realm of physical devices and the App Store, the security gets cranked up. In short, every app deployed to a physical device must be capable of being absolutely (and without a doubt) verifiable with a digital signature.

DEFINITION A digital signature is a mathematically-proven mechanism for showing the authenticity of digital documents, codes, or binaries. When a digital signature is applied to something, it is often referred to as having been *digitally signed*.

All iOS apps installed on a physical device are digitally signed, meaning that users of the app (and Apple) have reason to believe the app is authorized for installation by Apple and the app creator. Furthermore, because apps are digitally signed, users can be assured the app has not been tampered with on the device.

Let's get back to the question we posed earlier in this chapter, "Why do we need Xcode?". Aside from using Xcode to build your app, you also need Xcode to manage Apple's digital signature process for your apps. We won't lie to you: it's confusing, but we'll guide you through the confusion.

IOS APP SECURITY COMPONENTS

You need to become familiar with the following five iOS app security components:

3. Apple Developer Account associated with an Apple ID
4. App identifiers
5. Certificates
6. Registered devices
7. Provisioning profiles

WARNING At the surface, these components may seem simple, but there's a lot to explore and learn. In fact, there are entire books written on topics related to certificates alone. We are going to cover only the basics of these components, so we strongly encourage you to read Apple's official documentation before going any further:
<https://developer.apple.com/library/content/documentation/IDEs/Conceptual/AppDistributionGuide/MaintainingCertificates/MaintainingCertificates.html>.

APPLE DEVELOPER ACCOUNT ASSOCIATED WITH AN APPLE ID

Just about every service you use online requires that you have an account to log in and access special privileges. Apple is no different. You'll need an Apple developer account that is linked to an Apple ID to install apps on to an iOS device and submit apps to the App Store.

DEFINITION An Apple ID is an Apple-specific account used to log in to all Apple services, like iTunes, the Developer Member Center, the Apple Store, and so on. You can register for an Apple ID by going to <https://www.apple.com>. When you register, you'll need to provide an email address and a password. After the Apple ID is created, you'll refer to the Apple ID by the email address you used during registration.

DEFINITION An Apple developer account is an extension of your Apple ID that gives you the permission to install apps onto an Apple device and submit an app to the App Store. To create a developer account, go to Apple's developer member center at <https://developer.apple.com/membercenter> and login with your Apple ID.

There are two types of Apple developer accounts: a free account and a paid account. With a free developer account, you're able to install iOS apps on to devices through a process known as *sideload*.

DEFINITION Sideload is the process of using Xcode to install an iOS app directly to a physical device, without downloading it from the App Store.

To publish an app through the App Store, you'll need a paid developer account. Paid developer accounts cost \$99 a year. But wait. Don't let the \$99 fee scare you away. Everything you'll see in this chapter can be done with the free Apple developer account.

NOTE Apple has a program called the iOS Developer University Program, that allows universities teaching iOS development to allow students to enroll; however, if you wish to publish an app to the App Store, you must have a paid developer account. No discounts are offered currently. For more information on the iOS Developer University Program, see <https://developer.apple.com/support/university/>.

TIP We're not going to walk through the details of creating a developer account and Apple ID, but 9 to 5 Mac has a good step-by-step article on how to register: <https://9to5mac.com/2016/03/27/how-to-create-free-apple-developer-account-sideload-apps/>.

APP IDENTIFIERS

Back in chapter 3, you learned about the structure of your app's `package.json` file. Within the `package.json` file, is an `id` field, which specifies a unique identifier for your app. This unique identifier is referred to as an app's identifier (or `id` for short), and is formatted in reverse domain name notation.

NOTE You may recall the app Mike created for his sons, called My Robot, which had an app id of `com.brosteins.myrobot`. The Pet Scrapbook's app id is `com.brosteins.petscrapbook`.

Every iOS app must have a unique app ID before being installed onto an iOS device. You also need to register your app's ID in the Apple developer member center before the app can be installed on a device.

While we're on the topic of app IDs, let's use this opportunity to register the Pet Scrapbook in the developer member center. Start by browsing to the developer member center at

<https://developer.apple.com/membercenter> and logging in with your Apple ID. You should see several options, like figure 14.1.

NOTE As of early 2017, the figures throughout this chapter were accurate, but your screen may not look the same because Apple may have changed the developer member center website. If you're stuck, reach out in the book forums and someone will be able to help you. You can find the NativeScript in Action forums at <https://forums.manning.com/forums/nativescript-in-action>. Another great forum to check out is the official NativeScript forums at <https://discourse.nativescript.org>.

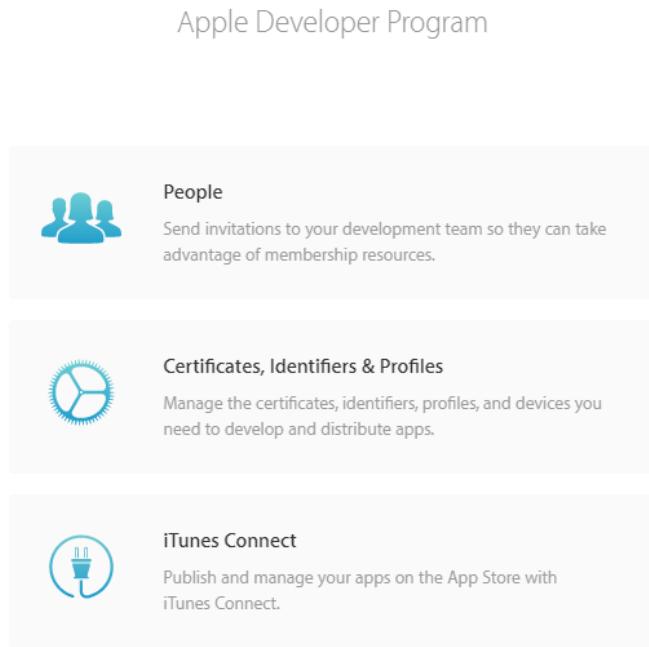


Figure 14.1 The Apple developer member center home page.

Click the Certificates, Identifiers & Profiles link to navigate to the Certificates, Identifiers & Profiles page.

DEFINITION The Certificates, Identifiers & Profiles page is where you register iOS app IDs and perform various other functions pertaining to app security.

You'll spend a lot of time on Certificates, Identifiers & Profiles page, and it can be overwhelming at first. But it's ok, we'll walk you through what you need to know.

On the left side of the page, locate the Identifiers area and click the App IDs link (figure 14.2).

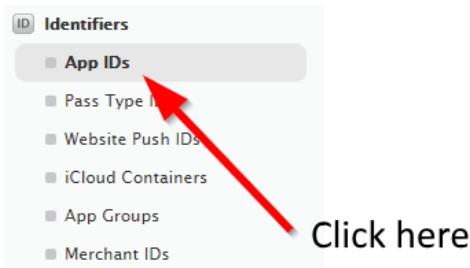


Figure 14.2 Navigate to the Identifiers areas by clicking the App IDs link.

The App IDs area lets you register new app IDs and view all your existing app IDs. Figure 14.3 shows our app IDs, plus two wildcard app IDs.

iOS App IDs	
14 App IDs total.	
Name	ID
AS2FieldService AppManager Test	com.AZ2FieldService.appmanagerTest
AppManager	com.brosteins.appmanager
BluegrassCellularRetailFinder	com.bigran.*
Brosteins Grocery	com.brosteins.grocery
Learn NativeScript	com.brosteins.LearnNativeScript
ManaCalc	org.nativescript.ManaCalc
NativeScript Demo App	com.brosteins.NativeScriptDemoApp
NativeScript in Action Pet Scrapbook	com.brosteins.petscrapbook
Pokedex	com.brosteins.pokedex
Robot Talk	com.brosteins.robottalk
SOTStatus	com.bigran.SOTStatus
Test	com.bigran.Test
XC Wildcard	*
Xcode iOS Wildcard App ID	*

Figure 14.3 The App IDs areas showing the various identifiers previously created and the + button to add a new ID. Note we have purposely obfuscated some of the identifiers.

DEFINITION Wildcard app IDs are special app IDs that allow Xcode to use a single, generic, app ID (*) to install any app on a device. We're not going to spend time discussing wildcard app IDs because we think it's important you understand how specific app IDs contribute to iOS security and publishing. If you'd like to learn more, Apple has a good support article at https://developer.apple.com/library/content/qa/qa1713/_index.html.

Let's add an app ID for the Pet Scrapbook by clicking on the plus button at the upper right of the App IDs screen (figure 14.3). As shown in figure 14.4, enter a description for the app ID in the *App ID Description Name* box, and enter the app's ID in the *App ID Suffix Bundle ID* box.

TIP You may end up with hundreds of app IDs over several years. Use the description to help you remember what app the app ID belongs to.

NOTE Our app ID is *com.brosteins.petscrapbook*, but if you're following along, you won't be able to register your version of the Pet Scrapbook with the same app ID. You'll have to create a different app ID. If you have your own domain name, you should use that as the prefix for your app ID. For example, *com.domainname.petscrapbook*. Otherwise, try *com.lastname.firstname.petscrapbook* for your app ID.

App ID Description

Name:
You cannot use special characters such as @, &, *, ', "

App ID Prefix

Value:  (Team ID)

App Name

App ID Suffix

① Explicit App ID

If you plan to incorporate app services such as Game Center, In-App Purchase, Data Protection, and iCloud, or want a provisioning profile unique to a single app, you must register an explicit App ID for your app.

App ID

To create an explicit App ID, enter a unique string in the Bundle ID field. This string should match the Bundle ID of your app.

Bundle ID:

We recommend using a reverse-domain name style string (i.e., com.domainname.appname). It cannot contain an asterisk (*).

Figure 14.4 Creating an App ID for the Pet Scrapbook by entering a name and the ID.

The App ID registration screen has several options for you to select (and a lot to read). We encourage you to read through the details, because it can give you a greater understanding of the App ID registration process. But, we'll summarize the important parts. We've already looked at the App ID Description section. There are additional options in the App ID Suffix section relating to wildcard app IDs.

WARNING You can create app IDs in the format of *com.brosteins.**, and use that app ID for multiple apps. But, when an app uses a wildcard app ID, it has restrictions like not being able to use Apple Pay, iCloud (Apple's cloud network), or Push Notifications. Your app may not need these services, but most apps we write tend to use at least one of these services.

The final section of the App ID registration page is the App Services section (figure 14.5).

App Services

Select the services you would like to enable in your app. You can edit your choices after this App ID has been registered.

- Enable Services:
- App Groups
 - Apple Pay
 - Associated Domains

Figure 14.5 The App Services section is the final section of the App ID creation screen, allowing you to select the services your app will have enabled.

The App Services section allows you to select from a variety of iOS and Apple services you would like to enable in your app. We're not going to cover what these services do in this book, but if you're interested, you can learn more at <https://developer.apple.com/library/content/documentation/IDEs/Conceptual/AppStoreDistributionTutorial/AddingCapabilities/AddingCapabilities.html>.

To add the app ID, click the *Continue* button, which will bring you to a confirmation page. Click the *Register* button to register your app ID. Navigating back to the Identifiers page now shows the Pet Scrapbook's app ID registered with Apple (figure 14.6).

iOS App IDs	
Name	ID
NativeScript in Action Test	ios.brosteins.nativescriptinaction-test
NativeScript in Action	ios.brosteins.nativescriptinaction
NativeScript in Action - Staging	ios.brosteins.nativescriptinaction-staging
NativeScript in Action - Dev	ios.brosteins.nativescriptinaction-dev
NativeScript in Action - QA	ios.brosteins.nativescriptinaction-qa
NativeScript in Action - Staging - QA	ios.brosteins.nativescriptinaction-staging-qa
NativeScript in Action Pet Scrapbook	com.brosteins.petscrapbook

Figure 14.6 The App ID screen showing the newly-created Pet Scrapbook app ID.

CERTIFICATES

Before we dive in to certificates, we hope you've read Apple's official documentation at <https://developer.apple.com/library/content/documentation/IDEs/Conceptual/AppDistributionGuide/MaintainingCertificates/MaintainingCertificates.html>. If you haven't, don't worry about it because it's a tough (and long read). We're going to give you the basics you'll need to get by with NativeScript iOS apps. If you're looking for a deeper dive after our explanation, don't let us hold you back.

DEFINITION Certificates are used in the digital signature process to prove and verify the creator of an app.

We're not going to go into detail about certificates because we don't have the space; it's relevant, but you can get by without knowing the intimate details.

Beyond the concept of certificates and that they're used to digitally sign apps, there are several types of certificates. We'll be covering two types that you'll need to get started: Development certificates (iOS App Development) and Production certificates (App Store and Ad Hoc).

Development certificates are used while you're developing an app and can be used to install apps on a device for testing purposes.

WARNING You're allowed to have one development certificate at a time, and the certificates are good for a year only.

Development certificates are meant to be a short-term solution used sparingly during your dev/test cycles. So, that leads us to Production certificates.

Production certificates are like Development certificates, but they have a longer life-span and can be used to publish apps to the App Store. Unlike Development certificates, you can have multiple Production certificates (but you're still limited to a total of three).

CREATING CERTIFICATES

Let's head back to the Certificates, Identifiers & Profiles page to learn how to create a certificate for the Pet Scrapbook. Navigate to the Certificates section by clicking the All link at the left, as shown in figure 14.7.



Figure 14.7 Click the All link to navigate to the Certificates section.

This page lists the various certificates you have created. When you come to the page for the first time, you won't have any certificates and you'll be prompted to get started (figure 14.8).

A screenshot of a web-based guide titled 'Getting Started with iOS Certificates'. At the top, there is a decorative icon of a blue-bordered certificate with a yellow seal. Below the icon, the title 'Getting Started with iOS Certificates' is centered. The page contains several sections of text and links:

- Request Certificates with Xcode**: Text explaining that Xcode is the easiest way to request certificates using the Organizer window.
- Request Certificates Manually**: Text instructing users to upload a 'Certificate Signing Request' from their Mac.
- Learn More**: A link to the 'App Distribution Guide'.

Figure 14.8 The page you'll see if you don't have any certificates.

There is a lengthy (and manual) certificate creation process you can go through in the developer member center. But, it's a waste of your time. Instead, we recommend doing it the easy way: use Xcode!

TIP Don't waste your time creating certificates in the developer member center manually. Use Xcode. It's easy!

WARNING Apple does a poor job of naming things consistently. You'll see certificates called *signing identities* in Xcode in some places, and *certificates* in others. You can ignore the different words and think of them both as certificates.

We're going to focus on using Xcode to create a certificate (or signing identity) for us. Let's get started by opening Xcode and creating a Development and Production certificate.

NOTE We're going to assume you already have Xcode installed, because it was a pre-requisite to running NativeScript on a Mac.

Open Xcode and go to the Xcode-Preferences menu (figure 14.9). If you've never launched Xcode, you'll be able to find it in your Applications folder.

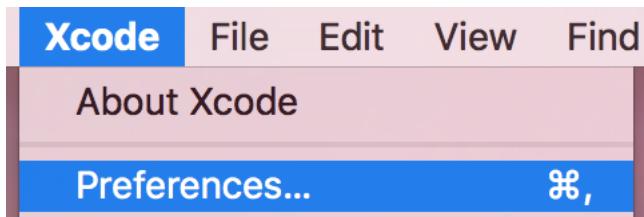


Figure 14.9 The Xcode – Preferences menu location.

From the Preferences menu, navigate to the Accounts tab and add your Apple ID using the plus icon at the bottom left. Figure 14.10 shows the plus button location and Mike's Apple ID that has been added.

NOTE This should be the same Apple ID associated with an Apple developer account.

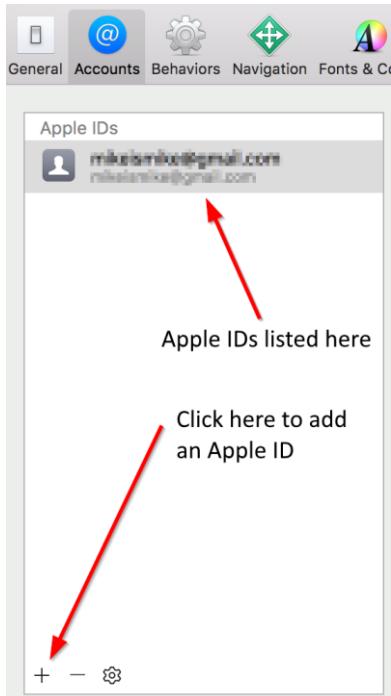


Figure 14.10 The Accounts tab showing the plus symbol to add an Apple ID and Mike's Apple ID added to Xcode.

Select the Apple ID and click the View Details button in the lower right of the Accounts tab, which brings up a page showing the details of your Apple developer account (figure 14.11).

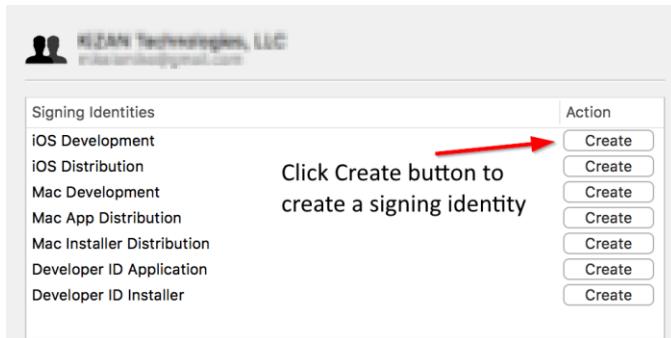


Figure 14.11 The Signing Identities section of the details page, showing the various certificate types that can be created.

At the top of the details page is a list of Signing Identities (a different name for certificates). To create a signing identity (or certificate), click the Create button. We'll be creating both a Development certificate (labeled iOS Development) and a Production certificate (labeled iOS Distribution).

WARNING Apple stinks at naming things consistently. A *Development certificate* is another word for iOS Development certificate and an iOS App Development certificate. Similarly, a *Production certificate* is another word for an iOS Distribution signing identity and an App Store and Ad Hoc Production certificate.

Create a Development certificate by clicking the Create button next to the iOS Development signing identity and wait. After about 15 seconds, the Create button disappears, meaning that a Development certificate was created.

NOTE Wait a second. I thought Apple's UX was epic? Somehow, disappearing buttons leave me wanting more.

If you check back on the Apple developer member center, a new Development certificate appears in the Certificates area (figure 14.12).

iOS Certificates		
1 Certificates Total		
Name	Type	Expires
Michael Branstein	iOS Development	Mar 11, 2018

Figure 14.12 The Certificates area showing the newly-created development certificate.

Wow. That was easy! Let's do the same for our Production certificate. Click the Create button next to the iOS Distribution signing identity, then refresh the Certificates page to see the Production certificate added (figure 14.13).

iOS Certificates		
2 Certificates Total		
Name	Type	Expires
[REDACTED]	iOS Distribution	Mar 11, 2018
Michael Branstein	iOS Development	Mar 11, 2018

Figure 14.13 The Certificates area showing the development certificate and the newly-created production certificate. Note the certificate name has been obfuscated on purpose.

NOTE You'll notice right away that the Production certificate has a different name than the Development certificate. But why? That's because Development certificates are for individual developers (and are named for the developer) and Production certificates can be shared across a team of developers (and are named for the team).

REGISTERED DEVICES

The fourth security-related component is *registered devices*. During the development and testing process for an app, you'll want to get your app in the hands of another person. The best way to do this is to sideload your app onto a device (iPhone, iPad, and so on). But, before you can sideload, you need to register the device's unique device identifier (UDID) with Apple.

DEFINITION A unique device identifier (UDID) is a 40-character string assigned to some Apple devices (iPhones, iPads, and so on). As the name suggests, these strings are unique and assigned by Apple at the time the device is created.

Registering a device's UDID is easy, but getting the UDID can be cumbersome. If you have access to the device and have a Mac, it's rather straightforward. Plug the device in via USB, open a Terminal session, and run the following command:

```
instruments -s devices
```

This command lists all physical and simulator devices attached to the Mac. Figure 14.14 shows the results of running this command on Mike's Mac.

```
mikeb-macbook-pro:~ mike$ instruments -s devices
Known Devices:
Michael's MacBook Pro [A7HES9A9A-5CA0-5D37-8779-43E880759468]
Brosteins (10.2.1) [29386a94e33ad834d5ea98cb3fb4280af63ce58e] UDID
Apple TV 1080p (10.1) [72394C09-007C-4749-B0E1-E970F5619770] (Simulator)
Apple Watch - 38mm (3.1) [A9A88988-7D40-4384-91C8-0976C9F4B604] (Simulator)
Apple Watch - 42mm (3.1) [D900E15AC3-8C09-4560-B1A9-10F802C304C8] (Simulator)
```

Figure 14.14 Output of the instruments -s devices command, showing the UDID of Mike's iPhone.

The UDID is the long 40-character string that we've obfuscated.

NOTE Yes *long sigh*, Mike's iPhone is named *Brosteins*. So, if you're ever in an airport and see a *Brosteins* WiFi hotspot, chances are, Mike's nearby.

Let's use this UDID to register Mike's iPhone in the developer member center. Navigate to the Devices area by clicking the All link (figure 14.15).



Figure 14.15 Clicking the All link navigates you to the Devices area.

Click the plus icon on the Devices page, and type the device name and UDID on the *Registering a New Device or Multiple Devices* page (figure 14.16).

Registering a New Device or Multiple Devices

Pre-Release Software Reminder

You may only share Apple pre-release software with employees, contractors, and members of your organization who are registered as Apple developers and have a demonstrable need to know or use Apple software to develop and test applications on your behalf.

Unauthorized distribution of Apple confidential information (including pre-release software) is prohibited and may result in the termination of your Apple Developer Program. It may also subject you to civil and criminal liability.

Enter a device name

Register Device

Name your device and enter its Unique Device Identifier (UDID).

Name:

Enter
UDID

UDID:

Figure 14.16 The device registration page, entering a device name and UDID.

TIP If you're going to be testing your app on multiple devices, you can upload a file containing the devices you wish to register instead of typing in each device individually. For more information on how to register multiple devices, check out Apple's official documentation at <https://developer.apple.com/library/content/documentation/IDEs/Conceptual/AppDistributionGuide/MaintainingProfiles/MaintainingProfiles.html>.

WARNING You can register only a maximum of 100 devices. This isn't necessarily a problem because you shouldn't need to test your apps on more than 100 devices.

PROVISIONING PROFILES

Ok! You've made it this far, and we thank you for that because it's been a long journey. But it's about to pay off. You've learned about the Apple developer account, Apple ID, member center, app IDs, certificates, and registered devices. But so far, these seem like they're very disparate and unrelated notions. *Provisioning profiles* tie these components together.

DEFINITION A provisioning profile is the glue that ties together a Development or Production certificate, app ID, developer, and registered device or devices. Once created, the provisioning profile is bundled with an app, authoritatively stating the app's ID, the creator, where the app can be installed, and various settings/services the app can use.

Just like certificates, there are two types of provisioning profiles: Development and Distribution. Development provisioning profiles are intended to be used with development certificates during the development and testing phases of your app's lifecycle. But, unlike certificates, provisioning profiles can change frequently!

NOTE Because provisioning profiles are a combination of certificates, app IDs, developers, and registered devices, any time one of these items changes, the provisioning profile must be changed. There are manual and automatic ways of handling this. We're going to teach you both, but think it's important that you understand the manual process before doing it automatically.

Development provisioning profiles change more frequently, as compared with Distribution provisioning profiles (especially the type that is specific to App Store Distribution).

Provisioning profiles are created in the Apple developer member center. Let's create a few for the Pet Scrapbook app. Navigate to the Provisioning Profiles section of the Certificates, Identifiers & Profiles page by clicking the All link (figure 14.17).



Figure 14.17 Navigate to the Provisioning Profile area by clicking on the All link.

Click the plus icon in the upper right of the iOS Provisioning Profiles page, and select the type of provisioning profile (figure 14.18).



What type of provisioning profile do you need?

Development

iOS App Development

Create a provisioning profile to install development apps on test devices.

tvOS App Development

Create a provisioning profile to install development apps on tvOS test devices.

Distribution

App Store

Create a distribution provisioning profile to submit your app to the App Store.

tvOS App Store

Create a distribution provisioning profile to submit your tvOS app to the App Store.

Ad Hoc

Create a distribution provisioning profile to install your app on a limited number of registered devices.

tvOS Ad Hoc

Create a distribution provisioning profile to install your app on a limited number of registered tvOS devices.

Figure 14.18 The provisioning profile page, showing options to create development and distribution profiles.

There are a lot of different provisioning profile types to choose from, but we're concerned with only three of them. These options are often confusing to new iOS developers, but we've broken it down to the following basics:

- Development – iOS App Development: use this type for all development and testing work. This profile allows your app to be installed on multiple devices that you specify.
- Distribution – App Store: use this type for a production release of your app to the App Store. This profile allows your app to be installed on all devices.
- Distribution – Ad Hoc: use this type of a production release of your app that *will not* be published in the App Store. This profile allows your app to be installed on multiple devices you specify. If you want to create a *private* App Store, this is the option to choose.

Let's start by creating a Development – iOS App Development provisioning profile, which allows us to do development and testing on a limited number of devices. Select *iOS App Development* and click the Continue button.

Select the NativeScript in Action Pet Scrapbook add id from the drop down (figure 14.19).

App ID: ▾

Figure 14.19 App ID selection as part of the provisioning profile creation process.

NOTE You'll recall we created this earlier in the chapter. If your app ID doesn't appear in the drop down, go back to the section about app identifiers and make sure you've added your app.

Next, select a Development certificate from the list of certificates (figure 14.20).

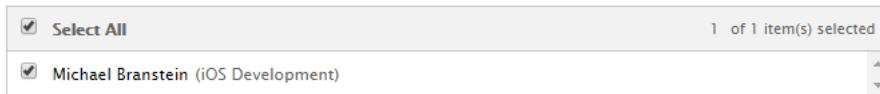


Figure 14.20 Development certificate selection as part of the provisioning profile creation process.

NOTE We created a Development certificate in the Certificates section of this chapter. If you weren't following along, go back and create a certificate in Xcode to continue.

Now select the registered devices you want to allow the Pet Scrapbook app to be installed on (figure 14.21).



Figure 14.21 Selecting Mike's iPhone from the list of registered devices as part of the provisioning profile creation process.

NOTE If your device doesn't appear in this list, go back to the Registered Devices section of this chapter to app your device.

On the final step, you'll name the provisioning profile (figure 14.22).

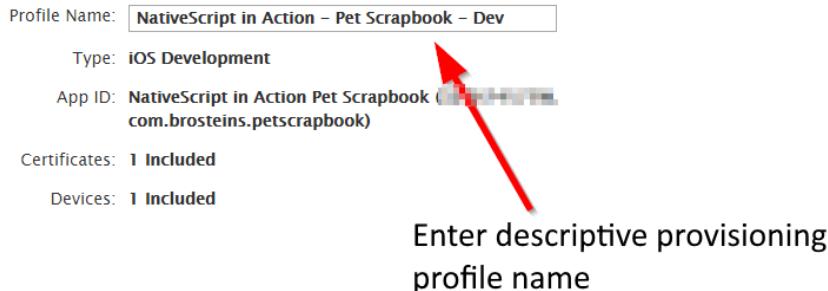


Figure 14.22 Provisioning profiles should be named in accordance with their app name, certificate, and purpose.

TIP Be descriptive in naming your provisioning profiles! Especially the development ones, because you'll end up with dozens as you add more and more devices and developers to your team.

After you create your provisioning profile, the last page of the provisioning profile creation wizard will present you with a link to download it. You're welcome to download the provisioning profile from the developer center website, but it's not necessary because there's an easier way to download and install the provisioning profile through Xcode. We'll show you how to use Xcode in a moment, so stick with us.

Before we continue, go ahead and create a second provisioning profile for the App Store, selecting Distribution – App Store as the type when creating it. When you’re finished, you should have two provisioning profiles for the Pet Scrapbook (figure 14.23).

Figure 14.23 The Provisioning Profile area, showing the newly-created profiles for development and distribution.

DOWNLOADING AND INSTALLING PROVISIONING PROFILES IN XCODE

Earlier in the chapter, we used the Account tab of the Xcode Preferences menu to create certificates automatically. Let's head back there and click the View Details button to open the details page for our developer account. Figure 14.24 shows this page.

Provisioning Profiles	Expires	Action
NativeScript in Action - Pet Scrapbook - App Store	3/11/18	Download
NativeScript in Action - Pet Scrapbook - Dev	3/11/18	Download

Figure 14.24 The Apple ID detail screen showing the provisioning profiles created online, with a Download button to install the profiles into Xcode easily.

At the bottom of the details page is a section labeled Provisioning Profiles. To download and install a provisioning profile, click the Download button next to it. After several seconds, the Download button disappears, meaning the provisioning profile has been downloaded and installed into Xcode.

TIP You'll know that you have downloaded and installed a provisioning profile when the *Download* link next to the provisioning profile disappears.

Phew. You made it! There's a lot to know and learn about security around your iOS apps, and unfortunately, you can't skip it. But, now that you have the basics under your belt, we're ready to use Xcode and publish the Pet Scrapbook!

14.1.2 Managing your app with Xcode

Earlier in this chapter, you learned how the NativeScript CLI transforms your app code (XML, JavaScript, and CSS) into an Xcode project. You also learned that the CLI could build that Xcode project into an iOS app without use the Xcode GUI, but we want you to become somewhat familiar with the Xcode GUI. Let's get started!

OPENING YOUR XCODE PROJECT

You may recall that the NativeScript CLI places the Xcode project into the *platforms* folder, but we never took a closer look. Figure 14.25 show the contents of the *platforms/ios* folder within the Pet Scrapbook app.

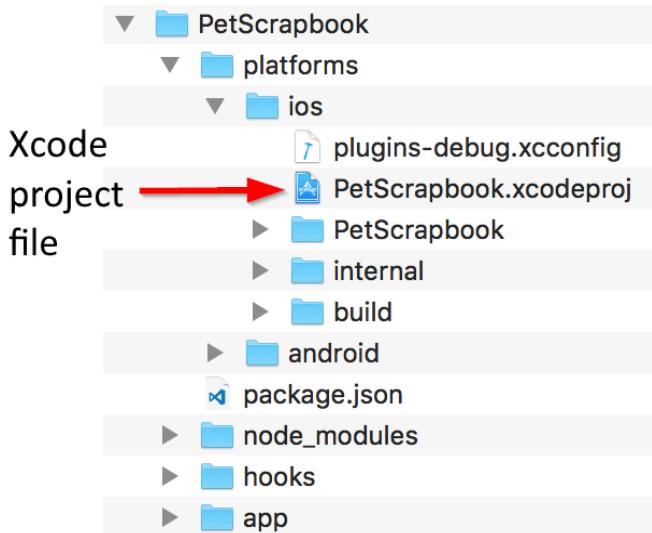


Figure 14.25 The platforms/ios folder contains the Xcode project for the Pet Scrapbook.

Within the *platforms/ios* folder, you'll find the *PetScrapbook.xcodeproj* file. This file is the Xcode project file. You can open the project by double-clicking this file, or by opening it from the *File* menu in Xcode.

If you're following along, let's open the Xcode project and investigate.

EXPLORING XCODE

After opening the Pet Scrapbook project, you're greeted with a busy UI, which on the best day is confusing. In all honesty, the first time we used Xcode, we struggled. A lot. Hopefully some of our tips will help you get started faster.

Start by expanding the PetScrapbook folder in the left-most area (figure 14.26).

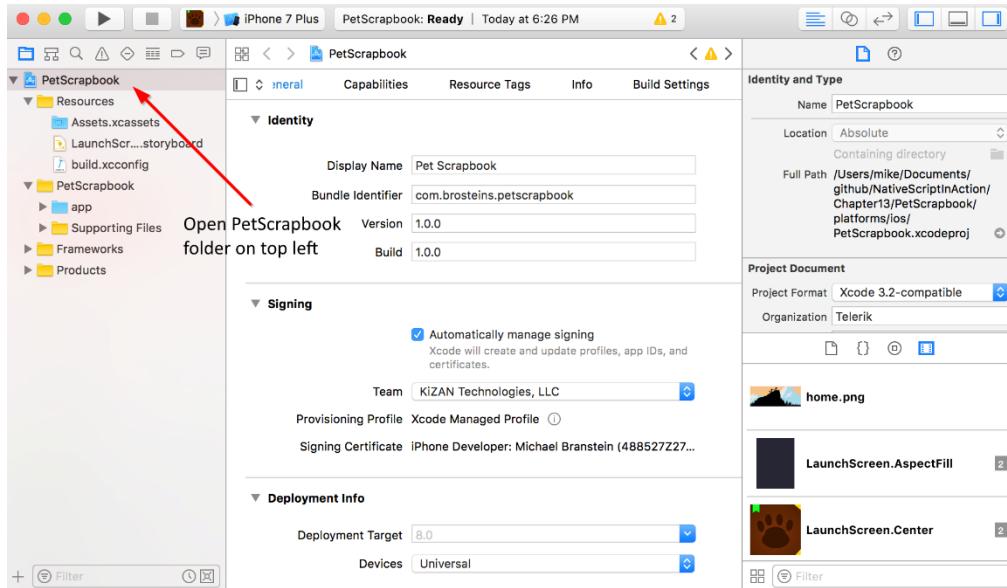


Figure 14.26 The Xcode GUI with the Pet Scrapbook project expanded open in the left-side navigator area.

The Xcode UI is separated into three areas. The left-most area is the navigator area, which shows all the files and folders that are part of the Xcode project. You'll quickly recognize several items under the PetScrapbook heading, including the Resources we created earlier in this chapter (Assets.xcassets), LaunchScreen.storyboard, and the build.xcconfig file. You'll also notice a copy of the PetScrapbook's app folder underneath the PetScrapbook folders. This is your app source code, copied by the CLI during the prepare process. It's nice to know the source code is there, but aside from that knowledge, you should ignore it.

WARNING Do not change files directly in the app folder you see in Xcode. Instead, modify only your app's original source code.

Selecting items in the left-most navigator panel will affect what's shown in the center detail area. The content of the center detail area changes, depending on the type of file or folder selected on the left. The exact content of the center detail area isn't important, but it's good to know that it will change as you click around on the left.

The area on the right contains two sub-areas: the inspector area and the library area. You won't need these areas, so we're going to skip over them.

Before we continue, feel free to click through several of the files and folders in the navigator area. When you're ready, select the topmost *PetScrapbook* folder. Let's explore the center detail area, as shown in figure 14.27.

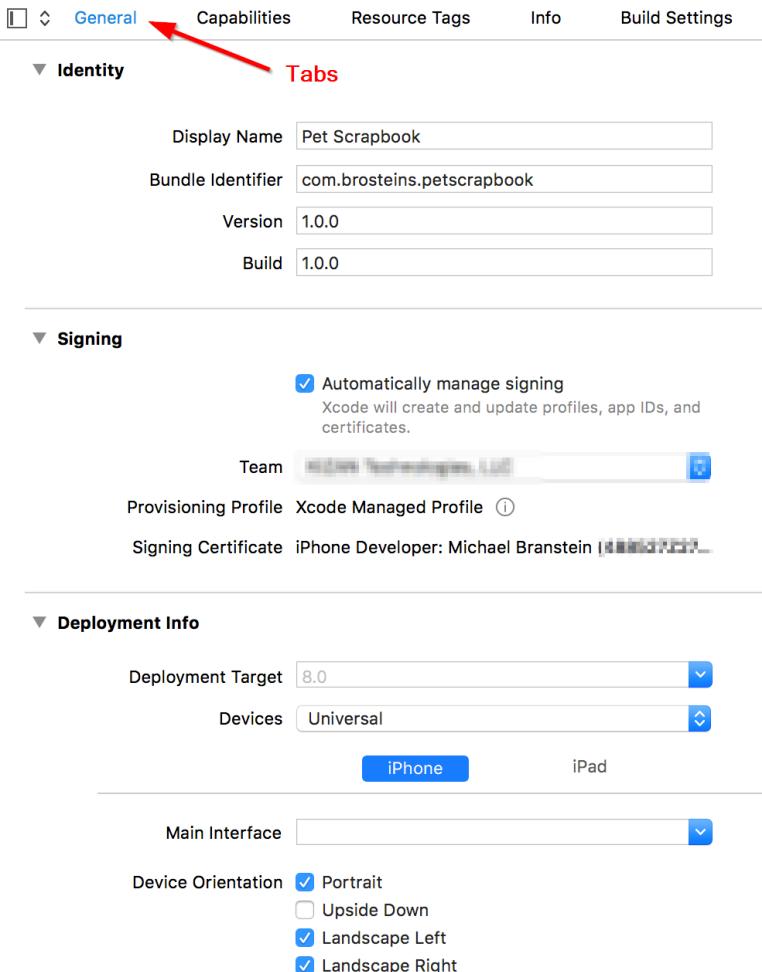


Figure 14.27 The center detail area, showing the General tab selected.

At the top of the central area are several tab pages to configure project-wide settings. You'll find tabs for general settings, app capabilities, and so on. There's a lot that can be configured, but the NativeScript CLI has done a lot of the heavy lifting for you, so we're just going to focus on the most important tab: *General*.

The General tab is separated into six vertical sections, as described in table 14.2. If you're feeling this is a bit overwhelming, we want to reassure you that you won't have to update more than one of these options directly.

Table 14.2 The six sections of the General settings tab in Xcode

Section	Description
Identity	This section configures the app's name, app ID, version, and build numbers. We've discussed these settings in detail throughout this chapter, so they shouldn't be surprising. As with most of the Xcode project, don't change these settings here. If you need to change them, do so in your app's source code.
Signing	This section configures which provisioning profile is used when building your app. We'll discuss this section in detail below.
Deployment Info	This section configures the iOS version your app targets, whether it targets iPhones, iPads, or both. It also configures which screen orientations your app supports. You'll recall these are all settings we previously configured in the <i>App_Resources</i> folder.
App Icons and Launch Images	This section configures which assets and resources your app will use to set the icons, launch images, or launch screen. Again, these settings were set by the CLI based on the changes we made to the <i>App_Resources</i> folder.
Embedded Binaries	This section contains references to the NativeScript core modules. You can ignore this section.
Linked Frameworks and Libraries	This section also contains references to the NativeScript core modules. Ignore this section as well.

You shouldn't directly change settings in these sections, but there is an exception we'll discuss next for the Signing section.

CONFIGURING APP SIGNING

The only section of the General tab you may want to change is the Signing section. As you'll recall, this section controls how digital signatures are applied to your app. This is controlled by the selected provisioning profile. Figure 14.28 shows the default settings for the Signing section.

▼ Signing

Automatically manage signing
Xcode will create and update profiles, app IDs, and certificates.

Team

Provisioning Profile

Signing Certificate

Status Signing for "PetScrapbook" requires a development team.
Select a development team in the project editor.

Figure 14.28 The Signing section of the General tab, showing the Team drop down.

The first thing we want to point out is a warning message that says, *Signing for "PetScrapbook" requires a development team.* This means we must select an option from the Team drop down to build our app in Xcode. Let's fix this issue by selecting a team. Figure 14.29 shows what's in our Team drop down.

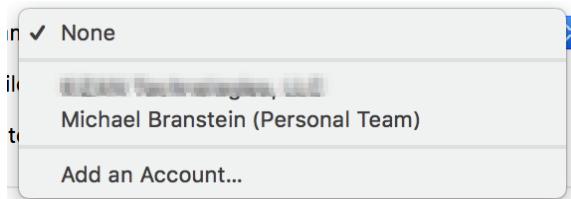


Figure 14.29 Mike's choices for selecting a team include our company's development team and Mike's personal team.

The Team drop down will list the various Apple developer accounts added to Xcode. Mike's personal developer account (Michael Branstein) is in the list, but he's also a member of our company's paid Apple developer account team (the obfuscated option).

NOTE The personal and corporate teams you belong to will differ from our teams.

We're going to select our company's paid Apple developer account team from the drop down, but that doesn't mean you need to do the same thing. But, how do you know which team you should pick?

TIP Picking the right team may seem confusing at first, until you think of it from the point of view of your provisioning profiles. Who owns the provisioning profiles you created earlier? Were they created as part of a team, or as part of your personal account? When you figure out who owns this app's provisioning profile, select that team.

After selecting our company's team, the warning message disappears (figure 14.30).

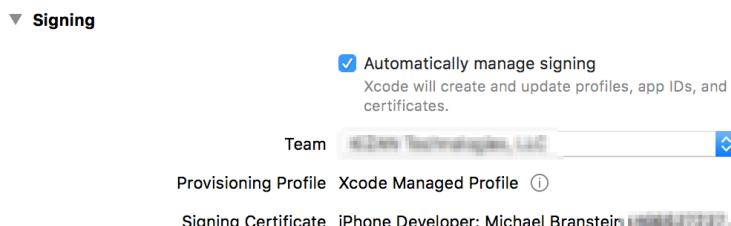


Figure 14.30 After selecting our company's team, the warning message disappears.

But wait! Something else odd is going on here. The signing certificate is set to Mike's developer certificate, but the provisioning profile isn't set to the profile we created earlier in this chapter. Instead, it's set to a profile named *Xcode Managed Profile*. And, what's that checkbox labeled *Automatically manage signing*? Seriously, what's going on?

NOTE Ok, the time has come. The whole cumbersome app ID, certificate, registered device, provisioning profile fiasco we went through isn't necessary when you're doing development and testing. Xcode can do all of that for you automatically, creating app IDs, certificates, and provisioning profiles. How nice. And convenient. But, please don't be angry with us. You needed to understand all those concepts before you got to this point. If you hadn't learned about app IDs, certificates, registered devices, and provisioning profiles already, we're not sure *how* we would have explained that all at this exact moment.

When the *Automatically manage signing* checkbox is checked, Xcode will do everything for you. But what about Production provisioning profiles? That's where we'll need to make some changes.

Even though it's convenient to have Xcode automatically manage app signing, let's uncheck the checkbox and manually configure Xcode to use the provisioning profiles we created earlier. When the checkbox is unchecked, two new sections appear. Figure 14.31 shows the *Signing (Debug)* and *Signing (Release)* sections.

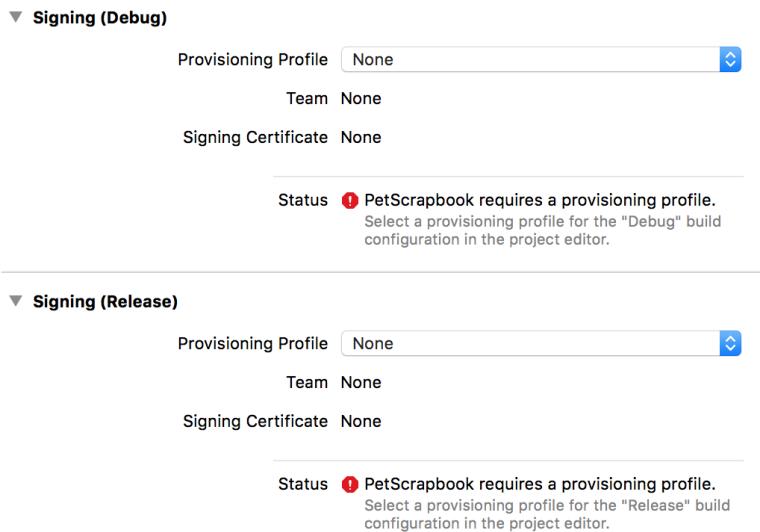


Figure 14.31 Unchecking the automatic signing management checkbox reveals sections for Debug and Release signing.

In both sections, a warning message is displayed, but you'll recognize it because it's the same message we've seen previously. To resolve the issue, select the appropriate provisioning profile for each section:

NativeScript in Action – Pet Scrapbook – Dev for Debug and NativeScript in Action – Pet Scrapbook – App Store for Release. After selecting the profiles, the team and signing certificate are set automatically (figure 14.32).

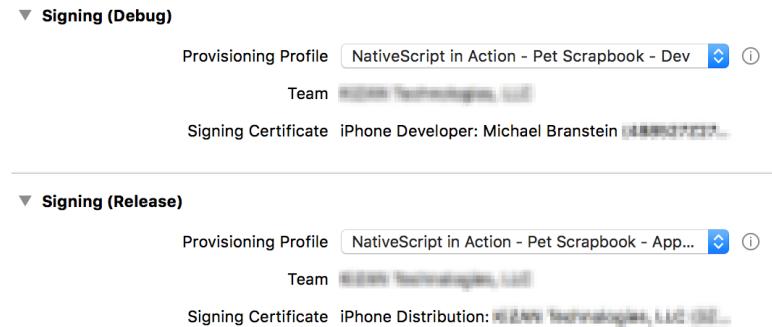


Figure 14.32 The Debug and Release Signing sections after selecting the development and distribution provisioning profiles.

And with that change, we're finished and ready to build our application in Xcode.

14.1.3 Building your app

Building a NativeScript app in Xcode is a two-step process:

1. Setting the active scheme to *Generic iOS Device*.
2. Running the Xcode build process.

SETTING THE ACTIVE SCHEME TO GENERIC iOS DEVICE

Before you start an Xcode build, the Xcode project's active scheme should be set to *Generic iOS Device*.

DEFINITION Xcode schemes are project-level configuration settings that define how an Xcode project will be built, the configuration settings that should be used during the build process, and a collection of tests to run. Xcode projects can have multiple schemes defined, but only one active scheme.

To select the active scheme, click the schemes selector in the upper-left corner of Xcode (figure 14.33).

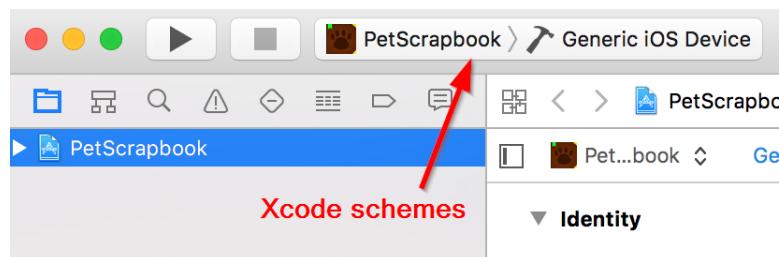


Figure 14.33 Xcode schemes can be changed by clicking the schemes drop-down at the top left of Xcode.

In the drop-down list that appears, scroll through the list until you find a section titled *Build Only Device*, and select the *Generic iOS Device* option (figure 14.34).

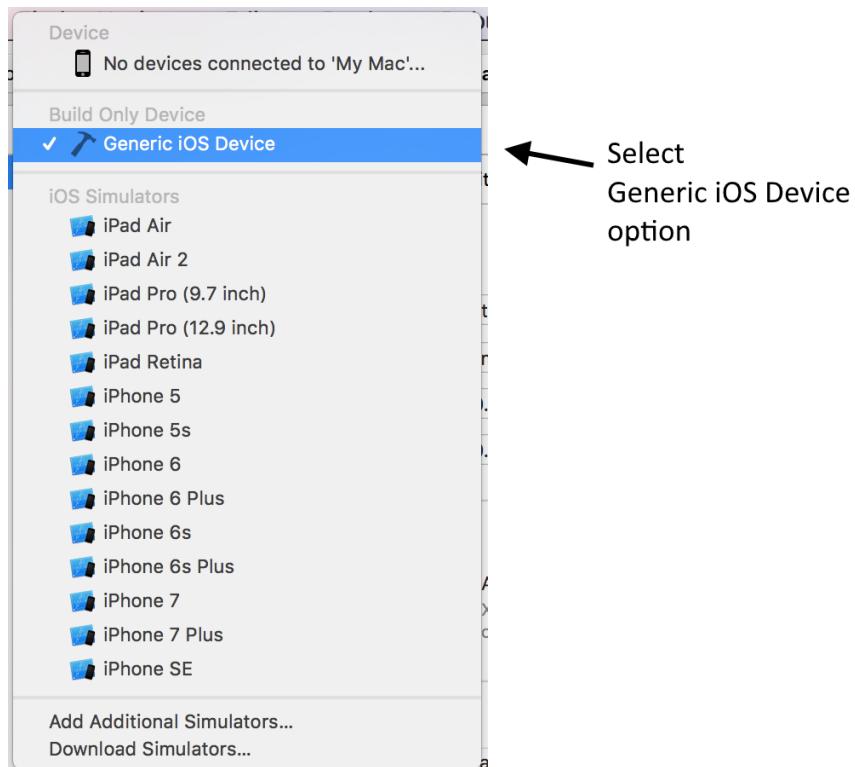


Figure 14.34 Select the Generic iOS Device scheme to set it as the active scheme.

RUNNING THE XCODE BUILD PROCESS

Starting the Xcode build process is by far the easiest thing we've done in this chapter. Navigate to Xcode's Product menu and select the Build option, as shown in figure 14.35.

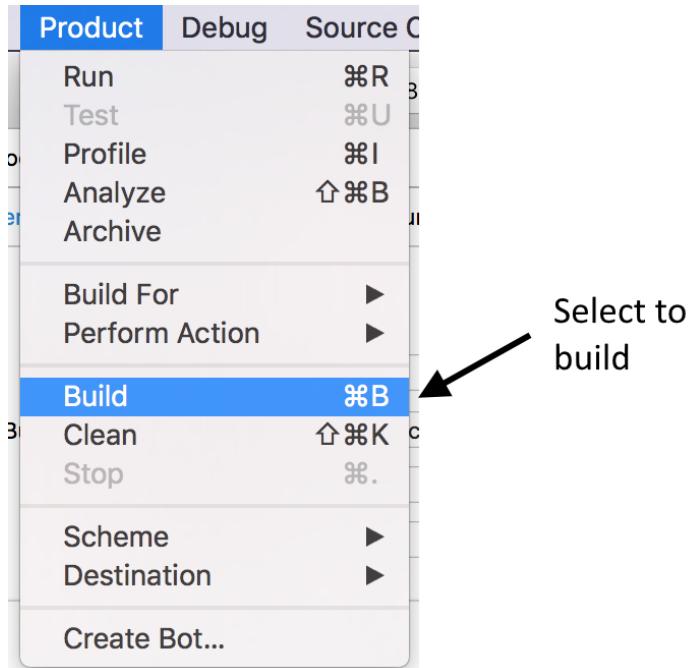


Figure 14.35 To Build an iOS app, go to the Product menu and select the Build option.

The build process can take anywhere from 5 to 60+ seconds, depending on the speed of your Mac, the size of your app, the number of images, and so on. When it's finished building, a build finished message will show.

14.1.4 Creating an app archive

The final step in getting your app ready for publishing to the App Store is to create an *app archive*. You'll recall that we mentioned archiving apps earlier in this chapter. Let's continue to explore app archives how they are created.

DEFINITION An archive is an .ipa file (which stands for iOS application archive), that includes the compiled binaries for an app. Archives submitted to the App Store are digitally signed with the certificate associated with the app's release provisioning profile.

Now that we have a build of the Pet Scrapbook, let's create an archive so we can upload it to the App Store.

Creating an archive is like building an app. From the *Product* menu, select the *Archive* option (figure 14.36).

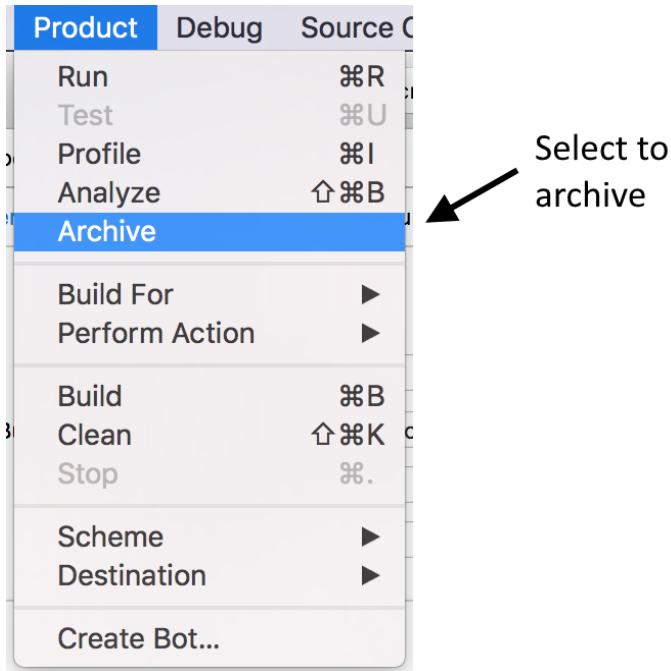


Figure 14.36 To create an app archive, select the Archive option from the Product menu.

When archiving an app, Xcode first re-builds the app, creates an archive, then opens the Organizer window (figure 14.37).

NOTE You may be wondering why you had to build the iOS app before archiving if archiving just re-builds it for you. It's a bit more in-depth than we want to go into in this book. Just accept that archiving your app will also re-build it first. If you're interested in some details and a discussion on building versus archiving, check out <http://stackoverflow.com/questions/14640816/how-to-create-xcode-archive-without-a-clean-build>.

DEFINITION The Xcode Organizer tracks the archives of iOS apps and allows you to upload an app archive to the App Store.

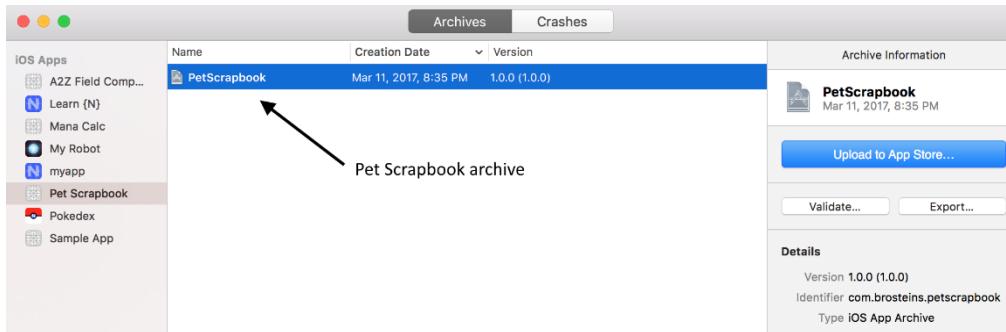


Figure 14.37 The Xcode Organizer showing the Pet Scrapbook app archive.

In the Organizer window, you'll find archives of all your apps that were built and archived by Xcode. The Organizer acts like a staging ground for your apps before being uploaded to the App Store.

UPLOADING YOUR APP TO THE APP STORE VIA ITUNES CONNECT

Now that you have an archive of the Pet Scrapbook app, the only thing left to do is upload the archive to the App Store via iTunes Connect (which you'll recall from the beginning of this chapter).

DEFINITION iTunes Connect is a section of the Apple developer member center portal dedicated to creating, preparing, and submitting apps to the App Store.

We're not going to walk you through the entire iTunes Connect experience of creating an app, uploading it, and so on for a couple of reasons:

1. It's quite time-consuming, and we don't have the space to give a lengthy explanation.
2. Using iTunes Connect is self-explanatory, because each step of the app creation and submission process is documented thoroughly on each step of the submission process.
3. There are several good walk-throughs available online, including
<https://www.raywenderlich.com/127936/submit-an-app-part-1> and
<http://codewithchris.com/submit-your-app-to-the-app-store>.
4. The App Store submission process changes regularly, and Apple's official documentation will be much more up-to-date than this book.

14.2 Summary

In this chapter, you learned the following:

- How the NativeScript CLI uses a two-phase process to create an Xcode project.
- How provisioning profiles are used to publish apps for development and production (App Store and Ad Hoc distribution) purposes.
- How to use the Xcode GUI to configure, build, and archive an iOS app.

14.3 Exercises

Using what you learned in this chapter, do the following:

1. Using the Apple developer member center, create a development and distribution provisioning profile for your app created in exercise 1.
2. Using Xcode, create an app archive using the provisioning profile created in exercise 2.

14.4 Solutions

1. To create a provisioning profile for the new app:
 - a. Go to the Apple developer member center.
 - b. Navigate to the Certificates, Identifiers, and Profiles area.
 - c. Create an app ID for the app named *org.nativescript.sampleapp*. Note that your app's id may be different.
 - d. Follow the instructions from the chapter section about certificates to create a development certificate if you don't have one already.
 - e. Follow the instructions from the chapter to add a registered device if you don't have any registered devices.
 - f. Go to the Provisioning Profile area and create a new development provisioning profile for the app ID, certificate, and registered devices created above. For specific step-by-step instructions, follow the walk-through in this chapter.
 - g. Do the same for an App Store distribution provisioning profile.
2. To create an app archive:
 - a. Run `tns build ios` from the command line to create an Xcode project.
 - b. Open the Xcode project from the *platforms/ios* folder of your app.
 - c. Select your project from the navigator's area in Xcode.
 - d. Uncheck the *Automatically manage signing* checkbox in the General tab of your Xcode project.
 - e. Select the provisioning profiles you created in the previous exercise for *Debug* and *Release* signing configurations.
 - f. Change the active scheme to *Generic iOS Device*.
 - g. Select the Archive option from the Products menu in Xcode.

Part 4: Angular and NativeScript

15

Creating a NativeScript App with Angular

This chapter covers:

- Why you may want to create NativeScript apps with Angular
- How to create and run a NativeScript-with-Angular app
- The structure of an Angular app

In the last chapters, we finished the Pet Scrapbook app and you learned how to prepare it for deployment to the App Store and Google Play store. We could stop here, and you'd have the tools and understanding needed to start writing your own apps with NativeScript. But, that wouldn't be fair because there's more to NativeScript than vanilla NativeScript. You can also create NativeScript apps with Angular, also known as *NativeScript-with-Angular apps*.

DEFINITION NativeScript-with-Angular apps are a type of NativeScript app that is written using the Angular JavaScript framework. Apps still rely on NativeScript's core modules and the NativeScript runtime, but let you replace NativeScript's app structure, page definition, navigation, and data binding with the corresponding Angular way of doing things.

Angular (a.k.a. Angular 2+) is the successor to the popular web development framework, Angular 1 (also known as AngularJS). Like NativeScript, Angular is an open source JavaScript framework, and it's maintained by Google and other development community members.

In this chapter, you'll learn why NativeScript apps with Angular are a compelling choice (versus vanilla NativeScript) and how to create NativeScript-with-Angular apps. Then, you'll wrap up the chapter by creating your first NativeScript-with-Angular app and comparing the structural differences between a vanilla NativeScript apps and a NativeScript-with-Angular app.

Before we jump in, we want to address another elephant in the room: what if you haven't used Angular before? Will this book teach you everything you need to write Angular apps? The short answer is no, and that's because writing apps with Angular can fill an entire book. Instead, you should know the basics of writing HTML apps with Angular. Then, we'll build on that knowledge and teach you how to use Angular and NativeScript together to write native mobile apps.

NOTE If you'd like to learn the basics of Angular, we highly recommend the official Angular QuickStart guide at <https://angular.io/docs/ts/latest/quickstart.html>.

If you're familiar with Angular, or just finished the QuickStart, keep reading. Otherwise, take a short detour and work through the Angular QuickStart at <https://angular.io/docs/ts/latest/quickstart.html>, then come back. From this point forward, we'll assume you have the basics of Angular under your belt.

15.1 Why Angular

So why are we talking about Angular in a NativeScript book? Most web developers have heard of Angular or have used it at least once. And, if you're looking to write native mobile apps, wouldn't it be great to build native mobile apps using the same frameworks and skills you use developing web apps? NativeScript with Angular does just that.

NOTE But wait, isn't Angular a JavaScript framework made for building web apps? Yes, it was. But as of v2.0.0, Angular was re-written to be platform-agnostic, meaning that Angular isn't about HTML, JavaScript, and CSS only. You can use Angular to write other types of apps, like NativeScript.

Skill reuse isn't the only advantage to using NativeScript with Angular over vanilla NativeScript. At the same time, NativeScript with Angular isn't all rainbows and unicorns: there are some disadvantages. Table 15.1 shows the different advantages and disadvantages of using NativeScript with Angular.

Table 15.1 Advantages and disadvantages of using NativeScript with Angular

Advantages	Disadvantages
No CSS bleed	Increased complexity
Passing data between pages is subjectively easier	Data binding syntax
Dynamic UI through Angular structural directives	TypeScript
Data binding	Filename conventions are not supported
TypeScript	Increased application size

NOTE You may have noticed that TypeScript is listed as both an advantage and disadvantage. This isn't a mistake, and we'll explain why below.

Let's take a closer look at the advantages and disadvantages of using Angular, then review our recommendations for when NativeScript with Angular makes sense.

15.1.1 Advantages

Throughout this chapter and the next, we'll go through an example of using NativeScript with Angular, and you'll learn about these advantages firsthand.

No CSS BLEED

If you've done web development in the past, you may love and hate CSS simultaneously. Its ability to cascade styles through an application is powerful: you can create global styles that can be overridden and adjusted on per-page and per-element basis. But, in larger applications, the cascading effect of CSS can be confusing and troublesome. For example, tracking down the source of a style that has been partially overridden multiple times is downright frustrating. Furthermore, selecting a "good" CSS class name can be hard: on large applications, you never know if someone else has used a class name you've selected. These same problems with CSS exist in NativeScript.

An advantage of using Angular is its handling of CSS by using something called *components*.

DEFINITION Angular components are a collection of UI elements and code that are walled-off from other components. By walled-off, we mean that components can't directly reach into each other. In a way, they have a force-field around them, limiting the data and interactions that come into the component and go out of the component.

A side-effect of the component model used by Angular is that CSS styles and classes do not traverse a component boundary. For example, assume there are two components, both with a defined class name `highlight`, but with different styles. Without components, the multiple definitions of the `highlight` class would bleed into each other. But, when components are used, both components can define a different `highlight` class and not worry about affecting other components.

PASSING DATA BETWEEN PAGES IS SUBJECTIVELY EASIER

Angular also provides some improvements to databinding by managing an observable on your behalf. Because Angular manages the `observables` object for us, it eases the process of accessing a single observable object across pages easy.

DYNAMIC UI THROUGH ANGULAR STRUCTURAL DIRECTIVES

Another great feature of Angular is its ability to create dynamic UIs by using *structural directives*.

DEFINITION Structural directives are parts of the templating engine of Angular that help to manipulate the layout of the UI. Some examples are structural directives that create FOR loops or IF-THEN statements inside a UI.

The most common structural directives, `NgIf`, `NgFor`, and `NgSwitch`, can be applied to any UI element, allowing you to dynamically add or remove UI elements via code written in the UI. In the next several chapters, we'll be using structural directives, and you'll learn how they can simplify the Pet Scrapbook's UI code.

DATA BINDING

Angular has rich support for data binding allowing you to integrate data into UI elements easily. Like vanilla NativeScript, NativeScript with Angular allows you to bind string expressions, events, CSS styles. We included data binding as an advantage because you might be familiar with the Angular data binding model. But, if you're not familiar, the syntax is comparable to NativeScript's.

TYPESCRIPT

The last advantage of using Angular in your NativeScript app is *TypeScript*.

DEFINITION TypeScript is an open-source development language maintained by Microsoft. TypeScript combines the features of JavaScript, but adds many of the capabilities of an object-oriented language (like classes and type-safety). Coding in TypeScript is like coding in JavaScript, but with a C# or Java-like syntax.

When creating an Angular app, you use TypeScript instead of JavaScript. We'll discuss TypeScript in detail later in this chapter, because it's important to understand. You may have noticed that TypeScript is also listed as a disadvantage to using Angular. This is because you may not have used TypeScript, and it may be a new skill you'll have to learn to write NativeScript-with-Angular apps. But, we believe learning TypeScript will be an essential skill for developers soon.

NOTE You can create vanilla NativeScript apps using TypeScript also, but we won't be covering that in this book because we think it's important to understand NativeScript by using JavaScript first. If you're interested in learning how to use TypeScript with NativeScript, check out <https://www.nativescript.org/using-typescript-with-nativescript-when-developing-mobile-apps>.

15.1.2 Disadvantages

We've looked at the advantages, but nothing's perfect in this world, and that includes NativeScript-with-Angular apps. Various advantages can be considered disadvantages for developers, because you need to first learn NativeScript, then learn Angular, and finally merge the two together. In this section, we'll explore our thoughts on why NativeScript-with-Angular apps may not be right for you.

INCREASED COMPLEXITY

NativeScript-with-Angular apps are more complex. They have more files than a vanilla NativeScript app and more npm dependencies. Angular is also a framework written for full-time developers, not the casual, write-a-website-on-the-weekend-for-your-relatives developer. Angular assumes you understand advanced software development concepts like dependency injection. Because of this assumption, if you're not already familiar with the advanced topics, it will take some extra effort to get started.

DEFINITION According to Wikipedia, *dependency injection* is a technique where one object supplies the dependencies of another object. Because this is an advanced topic in software development (and not important to this book), we're not going to cover this more. If you'd like to learn more, check out https://en.wikipedia.org/wiki/Dependency_injection.

All the items we just described make Angular more complex to understand, thus making it more complex to build a NativeScript-with-Angular app.

NOTE Don't let the increased complexity of Angular scare you away. Many developers we know prefer building NativeScript apps with Angular.

DATA BINDING SYNTAX

If you recall, we considered data binding an advantage because Angular data binding was like vanilla NativeScript data binding. The data binding syntax isn't a disadvantage technically. But, it's different. And that means you'll need to learn the Angular way of doing things.

TYPESCRIPT

Like data binding, TypeScript is both an advantage and disadvantage. We think TypeScript can be intimidating and possibly confusing for beginners. NativeScript and Angular make it easy to use, but because some syntax differs from JavaScript, we consider it a disadvantage for some.

FILENAME CONVENTIONS ARE NOT SUPPORTED

Back in chapter 2, we described how you can use the `.android` and `.ios` file-naming convention to have a file target a specific mobile platform. This is a powerful feature of vanilla NativeScript that can keep platform-specific code in separate files. Unfortunately, Angular doesn't respect this convention, so we consider it a disadvantage.

INCREASED APPLICATION SIZE

The most significant disadvantage is application size. Angular apps are much larger in size. The size difference is directly related to the size of Angular and the large number of dependencies. Furthermore, NativeScript-with-Angular apps have slow startup process, notably slower than vanilla NativeScript because of the large number of JavaScript files loaded. There is good news though. You can significantly reduce the size of your app while decreasing its size by using another tool called `webpack`.

DEFINITION Webpack is an open-source tool that can be used to reduce the size and loading time of JavaScript-based apps by *packing* (or bundling) the files in a smart way. Understand webpack and how it can be used in Angular apps is far beyond the scope of this book. For more information on webpack, check out <https://webpack.github.io> and the NativeScript with Angular webpack plugin at <https://www.npmjs.com/package/nativescript-dev-webpack>.

Because size and loading time of NativeScript-with-Angular apps is noticeably worse when compared to vanilla NativeScript, we recommend using webpack. Stay tuned, because we will cover it later in this book.

15.1.3 Recommendations

We've talked about the advantages and disadvantages of using Angular with NativeScript, but what does it really mean? A lot of the advantages and disadvantages are subjective because Angular has structural and syntactical differences that you may or may not prefer. And, when you use Angular with NativeScript, you're locked into doing it the Angular way.

Regardless, we're split: Nick prefers building NativeScript apps with Angular, and Mike prefers vanilla NativeScript. But, we do have an important recommendation for choosing Angular in a very specific circumstance. If you're writing a mobile app and think it may also be beneficial to be consumed as a web app, use Angular. You'll be able to reuse a large portion of your NativeScript code and build a web app that looks and behaves identically.

TIP If you're writing a mobile app and think it may also be beneficial to be consumed as a web app, use Angular. You'll be able to reuse a large portion of your NativeScript code, and build a web app that looks and behaves identically.

Our final recommendation is to learn NativeScript first, then learn Angular, then try them together. By learning NativeScript first, you'll get a solid NativeScript foundation and understand the boundaries between Angular and NativeScript.

15.2 Using NativeScript with Angular to recreate the Pet Scrapbook app

Now that you've learned about the advantages and disadvantages of creating NativeScript apps with Angular, let's dive in and create your first NativeScript-with-Angular app. Start by opening the command prompt on your computer, just like you do when you're creating a NativeScript app.

NOTE You may be wondering if you need to configure your computer in a special way, or install additional components to build a NativeScript-with-Angular app. We're happy to say, you don't. Like vanilla NativeScript apps, you use the NativeScript CLI to create, build, and run NativeScript-with-Angular apps.

Because you're familiar with the Pet Scrapbook app, we'll be using it as a reference point throughout the rest of the book. In this chapter, we'll start to create a NativeScript with Angular version of the Pet Scrapbook. By the end of the book, you'll have learned how Angular works with NativeScript and have a side-by-side comparison of the Pet Scrapbook written with and without Angular.

NOTE Before it gets confusing, let's talk about naming conventions for the two different versions of the Pet Scrapbook. We'll refer to the original vanilla NativeScript version of the Pet Scrapbook as the *original Pet Scrapbook*, or simply the *Pet Scrapbook*. When we're talking about the Angular version, you'll see the *Pet Scrapbook Angular*.

15.2.1 Scaffolding an Angular project

Earlier in the book, you learned to use the `tns create` CLI command to scaffold a vanilla NativeScript app. For example:

```
tns create PetScrapbook --template template-hello-world
```

Scaffolding a NativeScript-with-Angular app is the same as scaffolding a vanilla NativeScript app. You'll use the `tns create` command again, but with a different template. The NativeScript-with-Angular app template is named *nativescript-template-ng-tutorial*.

DEFINITION You might be wondering what the letters *ng* means. No, it doesn't stand for the *Next Generation* (think Star Trek). *Ng* is a common abbreviation for Angular. You'll see it throughout this book and in other places.

Let's start by creating an NativeScript with Angular version of the Pet Scrapbook. If you've been following along with the book, you'll already have created an app named *PetScrapbook*, so name the Angular version *PetScrapbookAngular*:

```
tns create PetScrapbookAngular --template nativescript-template-ng-tutorial
```

TIP You can also run `tns create PetScrapbookAngular --ng` to scaffold a NativeScript-with-Angular app, but we chose not to because it adds a lot of extra files you won't need. When using the `--ng` parameter, you cannot specify a template, because the CLI uses the default Angular template automatically.

Now that you've created the app, let's run it.

15.2.2 Running an Angular project

Running the Pet Scrapbook Angular app is the same as running the Pet Scrapbook app. Using the `tns platform add` CLI command, add the Android or iOS platform:

```
tns platform add ios  
tns platform add android
```

After adding the platform, run the app using the `tns run` command:

```
tns run ios --emulator  
tns run android --emulator
```

Figure 15.1 shows what the new Pet Scrapbook app looks like running on Android and iOS.



Figure 15.1 The new Angular Pet Scrapbook app running on Android and iOS, showing that it looks just like a vanilla NativeScript app.

The Pet Scrapbook Angular app looks and feels the same as a vanilla NativeScript app, but there's a lot more going on behind the scenes. Let's take a closer look, because although it's a NativeScript app, it's structured differently than a vanilla NativeScript app.

15.2.3 App structure

A difference between a NativeScript-with-Angular app and a vanilla NativeScript app is the structure of the app. In chapter 3, you learned about the structure of vanilla NativeScript app. Figure 15.2 shows the folder structure of the Pet Scrapbook Angular (left) compared with the Pet Scrapbook (right).

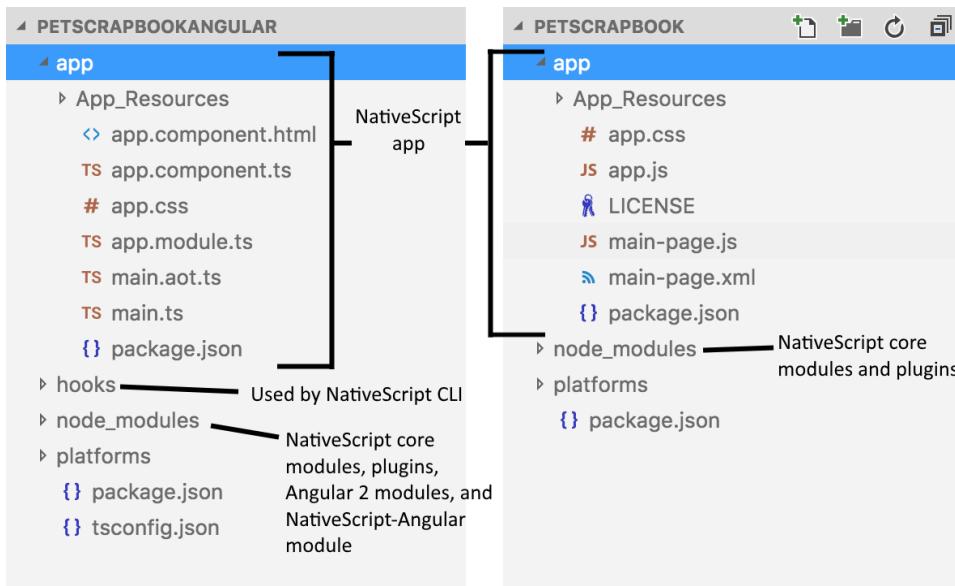


Figure 15.2 The project structure of the Pet Scrapbook Angular app compared with the structure of a vanilla NativeScript app.

At first glance, both projects look similar. Both apps have an *app* folder where the app code files are located.

You'll notice that many of the files are different. That's because this is an Angular app embedded within a NativeScript app, and the structure follows the structure of a standard Angular app, not a NativeScript app. We'll cover how Angular is embedded within a NativeScript app later in this chapter, but if you need a review of how an Angular app is structured, check out <https://angular.io/docs/ts/latest/quickstart.html>.

You may also notice the code files within the *app* folder don't end with .js. That's because the code files are TypeScript files, not JavaScript. We'll discuss TypeScript in greater detail shortly, so stay tuned. For now, just know that the .ts files are TypeScript files.

NOTE The `hooks` folder is used by the NativeScript CLI to help process the TypeScript files in the solution and compile them into JavaScript. We won't discuss these files because you don't need to modify them.

Finally, there are the `node_modules` and `platforms` folders, which you learned about in chapter 3. In NativeScript-with-Angular apps, these folders have the same purpose: the `node_modules` folder contains npm packages like the NativeScript core modules and plugins, the `platforms` folder contains the native apps for Android and iOS.

NOTE In a NativeScript-with-Angular app, the `node_modules` folder will have many more npm packages. That's because the app includes the Angular framework and other frameworks needed by NativeScript to work with Angular.

Now that you know about the difference in app structure, let's revisit the elephant in the room: TypeScript.

15.3 TypeScript

In the last section, you learned that TypeScript is an open-source development language, created and maintained by Microsoft.

NOTE You may be wondering why we're even talking about TypeScript. TypeScript is the preferred language for writing Angular code (per the creators of Angular). Because of this preference, the NativeScript with Angular templates assume you'll be writing your app with TypeScript.

Even though you're not required to write NativeScript-with-Angular apps using TypeScript, do yourself a favor by using TypeScript. We'll introduce you to the basics of TypeScript in this chapter, and help you write in TypeScript throughout the remainder of this book. If you're still on the fence about using TypeScript in your NativeScript-with-Angular app, you'll have to learn how to use JavaScript with Angular by going through the Angular JavaScript QuickStart at <https://angular.io/docs/js/latest/quickstart.html>.

15.3.1 Understanding TypeScript

TypeScript is a superset of JavaScript, meaning that all JavaScript code (and syntax) is TypeScript code. Figure 15.3 shows this relationship.

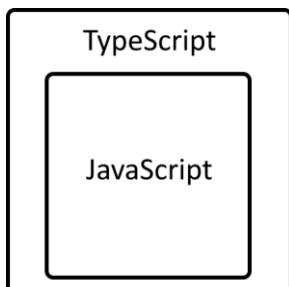


Figure 15.3 JavaScript code (and syntax) is a subset of the syntax that is valid in TypeScript.

At first, the relationship between JavaScript and TypeScript can be confusing. Think of it this way: if you write some JavaScript code like `var i = 1;`, it is both valid JavaScript and TypeScript. That's because TypeScript is based on JavaScript. But, if you write TypeScript-specific code, it isn't considered valid JavaScript.

Most browsers and JavaScript engines can't read TypeScript-specific code directly, so when you write TypeScript code, it needs to be transformed (or transpiled) into JavaScript code. Luckily, you don't need to do this compilation process manually: Microsoft provides you with a transpiler that can take a code file written in TypeScript and make an equivalent JavaScript file.

NOTE When you build your NativeScript-with-Angular app with the CLI, it transpiles your TypeScript code to JavaScript automatically. So, the whole compilation process is hidden from you. Because it's hidden and happens behind the scenes, we don't think it's important to dive deeper into the compilation process. For the purposes of this book, you should know it's happening, but not be concerned how. If you'd like to learn more, check out <http://www.typescriptlang.org>.

15.3.2 Why TypeScript is significant

If all TypeScript is just any valid JavaScript, then what makes it different? TypeScript is different because it enhances JavaScript by implementing object-oriented programming principles and modern programming language features of *ECMAScript*.

DEFINITION ECMAScript is the official specification name of programming languages such as JavaScript and TypeScript.

In our opinion, TypeScript is significant because it can be used as a *type-safe language*. JavaScript cannot.

DEFINITION A type-safe language is one that validates the data type of variables before operations are performed against them.

Defining variables in TypeScript works the same as JavaScript: `var firstName;`. But, you can go a step further in TypeScript and specify a variable's data type. For example, to define a string in TypeScript you would write `var firstName : string;`.

In addition to type safety, TypeScript also gives you the ability to define interface, class, Enums, generics, and many other object-oriented and modern language features. We won't be going into detail about every single feature of TypeScript because for the most part it will probably look very familiar to you as a JavaScript developer. If you'd like to review a detailed tutorial on TypeScript, go to <https://www.typescriptlang.org/docs/tutorial.html>.

15.4 NativeScript Angular integration

Earlier in this chapter, you learned that the `node_modules` folder contains many more npm packages in a NativeScript-with-Angular app. And, that's because of the additional of Angular and other dependent frameworks. In addition to the Angular-related packages, there's a special NativeScript-to-Angular integration package called NativeScript-Angular. Because NativeScript and Angular don't magically work together, the NativeScript-Angular packages works to marry them together.

NOTE The *NativeScript-Angular* packages are maintained by the NativeScript team. You can review the source on Github at <https://github.com/NativeScript/nativescript-angular>.

The *NativeScript-Angular* packages are just another npm package like the NativeScript core modules. You don't need to worry about adding these packages manually because they are added automatically when your NativeScript-with-Angular app is scaffolded (figure 15.4).

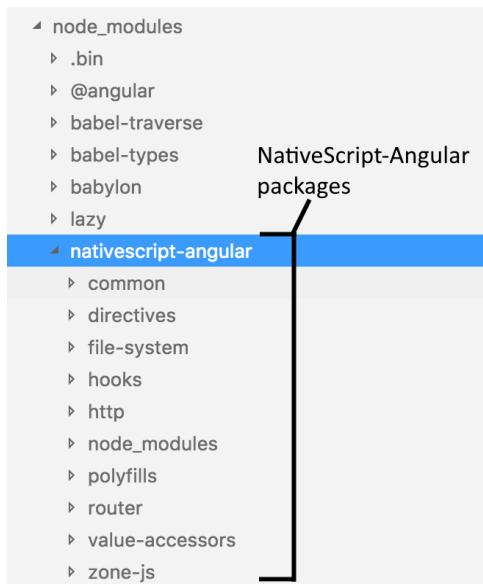


Figure 15.4 The `node_module` folder of the Angular Pet Scrapbook app contains various NativeScript-Angular packages. These are added to the project when it is scaffolded.

Let's start by reviewing how a NativeScript app using Angular app is launched.

15.5 Understanding NativeScript-with-Angular app startup

Now that you've learned about the structure and components of a NativeScript-with-Angular app (app structure, Angular and NativeScript-Angular packages, and TypeScript), let's see how they all work together. In chapter 3, you learned that the entry point for a NativeScript app is the `app.js` file. In a

NativeScript-with-Angular app, the entry point to the app is *main.ts*, as defined in the *package.json* file within the *app* folder. Listing 15.1 shows how the *main.js* file is set as the main entry point via the *main* property.

Listing 15.1 The main entry point defined in the package.json file of a NativeScript-with-Angular app

```
{
  "android": {
    "v8Flags": "-expose_gc"
  },
  "main": "main.js", // #A
  "name": "nativescript-template-ng-tutorial",
  "version": "1.0.1"
}
#A app entry point
```

If you're looking closely you may have noticed that the entry point of the app is *main.js*, but we said it was *main.ts*. This is a bit of semantics: yes, *main.js* and *main.ts* are different files, but remember, you're writing code in TypeScript. This means that you make changes to the *main.ts* file, and the NativeScript CLI transpiles it into the *main.js* file during the build process. You'll recall this happens behind the scenes, but it also means that the NativeScript runtime is really looking for the JavaScript files.

NOTE This is the only time you'll see us refer to a JavaScript file in the NativeScript-with-Angular app. Throughout the rest of the book, we'll be referring to the TypeScript files, because that's what you'll interact with. But understand that behind the scenes, NativeScript transpiles these files into JavaScript, and the JavaScript is what NativeScript really uses.

Now that you understand how the NativeScript runtime finds the *main.ts* file, let's see how it is used to start the app. Listing 15.2 shows *main.ts* from the NativeScript-with-Angular Pet Scrapbook app.

Listing 15.2 The main entry point to an Angular app main.ts

```
import { platformNativeScriptDynamic } from "nativescript-angular/platform"; // #A
import { AppModule } from "./app.module"; // #B
platformNativeScriptDynamic().bootstrapModule(AppModule); // #C
#A Import the module to launch an NativeScript app inside of Angular
#B The main settings of the Angular app
#C Launch the app
```

Listing 15.2 introduces a new concept from the TypeScript language: *imports*. In TypeScript, you can import a method, function, class from another file in your project. Importing a method is like the require syntax that we use to reference a core module in a vanilla NativeScript app.

NOTE The import statement automatically looks in the *node_modules* folder for references, just like the require statement does. When a package is imported, it is referenced via its file name without the *.ts* extension, but it really loads the file with the *.ts* extension.

At a high-level, the code in *main.ts* imports the NativeScript-Angular integration layer and an Angular module from the *app.module* file. Then, it launches the Angular runtime, referencing the module just imported. You don't need to know any further details of the code in *main.ts* because this is code that's provided for you via the NativeScript with Angular template.

15.5.1 Understanding the *app.module.ts* file

Let's take a closer look at the *app.module.ts* file, which is loaded by the Angular runtime when the app starts (listing 15.3).

Listing 15.3 The *app.module.ts* file showing the application settings

```
import { NgModule, NO_ERRORS_SCHEMA } from "@angular/core";
import { NativeScriptModule } from "nativescript-angular/platform"; // #A

import { AppComponent } from "./app.component";

@NgModule({
    declarations: [AppComponent], // #B
    bootstrap: [AppComponent], // #B
    imports: [NativeScriptModule], // #B
    schemas: [NO_ERRORS_SCHEMA], // #B
})
#A The main NativeScript module to connect NativeScript and Angular together
#B Create our Angular app with specified settings
```

As you continue to build the Angular Pet Scrapbook you'll periodically visit the *app.module.ts* to modify the settings of your app. If you've gone through the Angular tutorials on <http://angular.io>, you'll recognize the contents of the *app.module.ts* file. It's standard Angular module instantiation. We're not going to dive into the details of what an Angular module is, so check out <http://angular.io> if you're not sure what we're referring to.

For now, it's important to note a couple of things in the *app.module.ts* file. The *NativeScriptModule* import statement gets a reference to the main *NativeScript-Angular* library. Normally, an Angular app runs inside a web browser, but we are running the app inside of the NativeScript runtime, and various browser-specific things (like the document and window object) don't exist. Because of this, we'll pass *NativeScriptModule* into the *@NgModule* declaration.

@NgModule is a *TypeScript* decorator that helps define further metadata about the class immediately following the decorator: *AppModule*.

DEFINITION *TypeScript* decorators are a special type of function that can precede a class declaration, method, accessor, or property. They're called decorators because they're interpreted as decorating code that comes directly after it, and providing additional information (or metadata) about the code. Decorators use the @ syntax to define themselves, and are evaluated like a function that is run immediately before the execution of the code it decorates. To learn more about decorators, check out the official *TypeScript* documentation at <https://www.typescriptlang.org/docs/handbook/decorators.html>.

For the purposes of this book, you don't need to know much about decorators. Remember that `@NgModule` is like a function that runs right before the class `AppModule` is created, and (like a function) it has various parameters that can be passed in. The parameters include:

- The `bootstrap` parameter is the root *component* of the Angular app, meaning that when the app is launched, the `AppComponent` will be loaded.
- The `declarations` array tells Angular which *components* (or pages) are available to the app.

DEFINITION Angular components can be thought of as different pages of a NativeScript app. We're not going to explain how Angular components work in detail, because you should already know about them from running through the Angular tutorial at <http://angular.io>.

NOTE When a component is referenced in a NativeScript-with-Angular app, it is referenced via its file name without the `.ts` extension. For example, when `app.component` is referenced, it refers to the file `app.component.ts`.

When the `@NgModule` statement runs, `AppComponent` is loaded, which points to the `app.component` file. Let's look at `app.component` to finish our walk-through of NativeScript-with-Angular app startup.

15.5.2 Angular components and the `app.component.ts` file

Angular apps can be broken down into many different components, and NativeScript-with-Angular apps use Angular components heavily. In the previous section, you learned that components can be thought of various pages of a NativeScript-with-Angular app, but they're a bit more than just a page. Components are separate self-contained views that can be used once as a page, or reused as a sub-section of multiples pages. Later in this book, you'll see how you can reuse a component across several pages, but for now, let's stick to the idea that a component is a page.

The component defined in the `app.component.ts` file is the initial component (or page) loaded into a NativeScript-with-Angular app. You'll recall this happened via the `@NgModule` declaration in the `app.module` file. Listing 15.4 and figure 15.5 show the code definition of this component running in the Pet Scrapbook.

Listing 15.3 The `app.component.ts` file defining an Angular component

```
import { Component } from "@angular/core";

@Component({
  selector: "my-app",
  template: `
    <ActionBar title="My App"></ActionBar>
    <!-- Your UI components go here -->
  `
})
export class AppComponent {
  // Your TypeScript logic goes here
}

#A Component declaration creates an HTML tag named my-app that will render the contents of the template property
```

When this component is loaded, the `@Component` declarative statement creates a new Angular component named `my-app`, which when loaded will render the contents of the template (the Action Bar markup in listing 15.3, shown in figure 15.5).

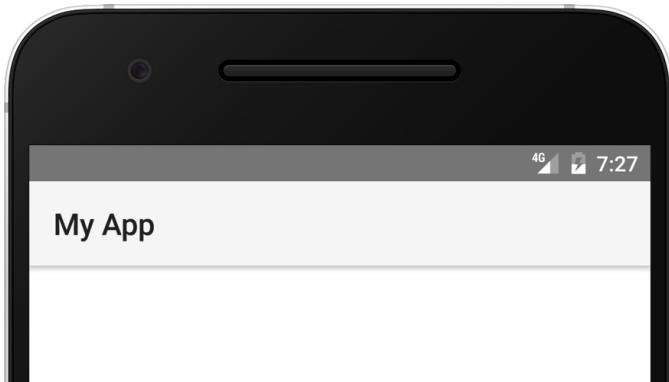


Figure 15.5 The component defined in `app.component.ts` of the Pet Scrapbook app.

The content of the template element in `app.component.ts` is the page that is loaded when the Pet Scrapbook Angular is launched: an action bar is placed on the page, showing the text `My App`.

It's straightforward to see how that the template's contents are displayed in the NativeScript app, but let's dissect the component further. Angular components have two major parts: metadata and a template.

COMPONENT METADATA

The metadata file, `app.component.ts`, defines the settings for a component and tells Angular how to process and use the component.

NOTE The metadata of a component is also defined using a declarative TypeScript statement-like module. You don't need to know the details about this now, because we'll take a closer look at creating components in the next chapter.

COMPONENT TEMPLATE

The template part of an Angular component defines the UI or view code. In a traditional Angular app that runs in a web browser, the template is just HTML code. In a NativeScript-with-Angular app, the template is NativeScript XML view components, which we discussed previously such as `StackLayout`, `Label`, and `Button` elements.

MOVING ON

And, that's it! Now you know how a NativeScript-with-Angular app starts up. It's a little different from a vanilla NativeScript app, but that's because we're using Angular. As we continue to build the Pet Scrapbook using Angular, we'll be creating components for each of the pages that we created in previous chapters like the home page and list view page.

If you're feeling overwhelmed or confused, that's ok. The first time we started using Angular, we felt overwhelmed too. We'll move slowly and guide you through each step, explaining along the way. And, if you're feeling that you need an Angular refresher, check out the Angular tutorial on <http://angular.io> or one of Manning's other books on Angular (like *Angular 2 Development with TypeScript* at <https://www.manning.com/books/angular-2-development-with-typescript>).

15.6 Summary

In this chapter, you learned:

- How to create a NativeScript-with-Angular app
- How the structure of a NativeScript-with-Angular app differs from a vanilla NativeScript app
- How TypeScript is used when creating a NativeScript-with-Angular app
- How NativeScript and Angular work together

15.7 Exercise

Create a new NativeScript-with-Angular app named *HelloAngular* using the *nativescript-template-ng-tutorial* template. Then, change the action bar to show *Hello Angular!*

15.8 Solutions

To create a new NativeScript-with-Angular app named *HelloAngular* using the *nativescript-template-ng-tutorial* template, use the NativeScript CLI:

```
tns create HelloAngular --template nativescript-template-ng-tutorial
```

To change the action bar to show *Hello Angular!*, modify the *app.component.ts* file template element (listing 15.4).

Listing 15.4 The *app.component.ts* file showing an action bar with the text *Hello Angular!*

```
import { Component } from "@angular/core";

@Component({
  selector: "my-app",
  template: `
    <ActionBar title="Hello Angular!"></ActionBar>
    <!-- Your UI components go here -->
  `
})
export class AppComponent {
  // Your TypeScript logic goes here
}
```

16

Using Angular components and routing

This chapter covers

- Creating Angular components
- Navigating between components with routing

In the last chapter, you learned how to create a NativeScript-with-Angular app and began to re-create the Pet Scrapbook. We'll be continuing to refine the Pet Scrapbook Angular by learning about Angular components and routing, two core concepts that parallel the concepts of vanilla NativeScript: pages and navigation.

In our experience, learning about Angular concepts on their own (without the context of a vanilla NativeScript app) can be confusing, so we've taken the time to build on things you've already learned about to help you quickly learn about Angular and how it's used in a NativeScript-with-Angular app. Just like the last chapter, if you haven't walked-through the official QuickStart guide for Angular, we strongly recommend you check it out at <https://angular.io/docs/ts/latest/quickstart.html>.

16.1 *Creating static components*

As you'll recall, you learned that the `app.module.ts` file is the entry point for NativeScript-with-Angular apps. You also learned that Angular components are like NativeScript pages, but let's dive a little deeper into how components and pages are similar. Figure 16.1 shows the contents of an Angular component.

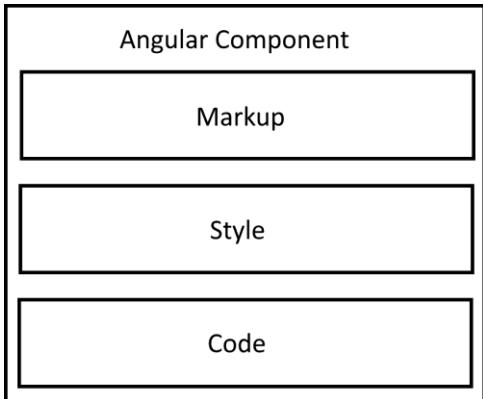


Figure 16.1 Angular components have markup, style, and code.

Angular components comprise of three pieces: markup, styling, and code. The Angular-specific pieces of a component are just like the vanilla NativeScript pieces of a page, but with implementation differences. Angular components are built with HTML markup, CSS styling, and TypeScript code, and vanilla NativeScript pages are built with XML markup, CSS styling, and JavaScript code.

NOTE You may have noticed that we said Angular code is written in TypeScript and NativeScript code is written in JavaScript, but that's not correct technically. Behind the scenes, all code that is run in a vanilla NativeScript and NativeScript-with-Angular app is JavaScript. But, when you're writing apps, most people choose to write NativeScript-with-Angular apps with TypeScript, and vanilla NativeScript apps with JavaScript. This trend is why we described components and pages with TypeScript and JavaScript.

Figure 16.2 shows the implementation differences between Angular components and NativeScript pages.



Figure 16.2 The differences between a NativeScript Page and Angular Component.

Writing the code for apps in different language variants isn't the only code difference: the structure and content of the code is different. The difference may be obvious to some, but we want to point it out

to ensure we're on the same page. The code within Angular components is Angular-specific, using Angular constructs like the `@Component` declaration we saw in chapter 15 and in listing 16.1.

Listing 16.1 The app.component.ts file showing Angular constructs like the `@Component` declaration

```
import { Component } from "@angular/core";

@Component({
  selector: "my-app",
  template: `
    <ActionBar title="Hello Angular!"></ActionBar>
    <!-- Your UI components go here -->
  `
})
export class AppComponent {
  // Your TypeScript logic goes here
}
```

There is a final difference between Angular components and NativeScript pages in how the view markup is written (HTML versus XML), and you'll learn about it shortly. But it's easier to teach you about the difference if we've built an Angular component first. So, let's tackle our first component (or page) of the Pet Scrapbook Angular: the Home component.

16.1.1 Home component

Let's start creating the Home component by adding the three files required for an Angular component. Create a folder named `views` in the `app` folder, then add a second folder named `home` to the `views` folder.

TIP Angular apps have their own style guide for organizing modules, components, and other Angular constructs. Per the official style guide, components should be placed in the `views` folder, and further organized by subfolders with the name of the view. We'll help you learn the basics of the Angular style guide, but you should check it out for the specifics: <https://angular.io/styleguide>.

Create three files in the `home` folder: `home.html`, `home.css`, and `home.component.ts`.

TIP Per the Angular style guide, the three component-related files should be named with the component's name and a `.html`, `.css`, and `.component.ts` extensions. The `.html` file contains the view markup, styling is in the `.css` file, and the `.component.ts` file contains the Angular-specific component declaration written in TypeScript.

Figure 16.3 shows the resulting Pet Scrapbook Angular solution with the new files.

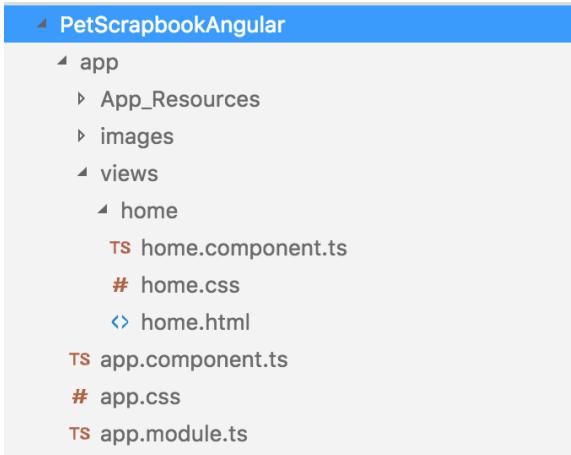


Figure 16.3 The Home component files added to the Pet Scrapbook solution.

After adding the Home component files to the Pet Scrapbook Angular app, we'll create the *Home* page using Angular. Figure 16.4 shows the *Home* page in case you don't remember.



Figure 16.4 The Home page of the Pet Scrapbook app that will be replicated using the Home component in Angular.

UPDATING THE VIEW MARKUP

Start by copying the XML view code from the Pet Scrapbook app and placing it in the *home.html* file. Listing 16.2 shows the contents of *home.html*.

NOTE You may have noticed that we made some changes to the markup after copying it to *home.html*. Specifically, we removed the page UI element (and we'll explain why later). We also removed the

databinding for the title and footer UI elements. We did this to simplify the Home component. Don't worry though, we'll introduce you to Angular data binding later in the chapter.

Listing 16.2 The home.html file showing the UI markup of the Home component

```
<ScrollView> // #A
  <StackLayout>
    <Label class="h1 text-center m-y-10" text="Pet Scrapbook"></Label> // #B
    <Image src("~/images/home.png")></Image>
    <Label class="footnote text-center m-y-10" text="Brosteins ©2016"></Label> // #B
  <StackLayout orientation="horizontal" horizontalAlignment="center">
    <Button class="btn btn-primary btn-rounded-sm btn-active m-r-20"
      text="About">
    </Button>
    <Button class="btn btn-primary btn-rounded-sm btn-active m-l-20"
      text="Continue">
    </Button>
  </StackLayout>
</StackLayout>
</ScrollView>

#A Page UI element removed from around scroll view
#B Data binding removed for simplicity
```

Earlier in this chapter, we mentioned there is another difference between Angular component view markup and vanilla NativeScript view markup. Angular component view code is a blend of HTML/XML NativeScript-iness markup, not XML. Huh? Yeah. If you're confused, we understand. We were confused at first too. Let us explain.

Angular is a generic framework for building apps, and it can run in all sorts of places. For example, Angular can run in the browser (Chrome, Firefox, Edge, and others) or a mobile app (like NativeScript). Regardless of where it runs, its view code must be written in a language and syntax that's recognizable by the underlying platform. This means that in a browser, the view code must be written in HTML (because browsers know how to parse and render HTML to a web page). In a NativeScript app, view code is written in HTML/XML-ish NativeScript syntax that a NativeScript app can parse and render into native app UI elements.

More specifically, Angular-component view code (in NativeScript-with-Angular apps) isn't HTML or XML, but it resembles both very closely. It's not web-based HTML markup because it doesn't use *h1*, *h2*, *h3*, *div*, *p*, *form*, and *input* HTML elements, and it's not exactly NativeScript XML because the data binding syntax used by Angular isn't XML-compliant.

So, what is it?

There's not a great name for the view markup used in NativeScript-with-Angular app, so let's call it what it is: markup.

NOTE The NativeScript community has discussed naming this special markup NativeScript HTML or NSML, but it never really felt right. Plus, calling it NSML would lead some to name the view file with an extension of *.nsml*, breaking code editor Intellisense support you get for free when you name your view files with an extension of *.html*.

Phew! That's a lot to take in and understand about a minor technicality, so if you're feeling lost, here's all you need to know about creating Angular component views in NativeScript:

- You write HTML/XML-like markup using NativeScript UI elements.
- Markup doesn't need to include the page UI element.
- Even though the markup isn't HTML, it's placed in a file with an extension of `.html` because many code editors have intellisense support for Angular apps that only works for files with an extension of `.html`.

ADDING CSS STYLES

Like vanilla NativeScript apps, NativeScript-with-Angular apps support both global and page-specific styling. The `app.css` file contains the global CSS definitions, and the component-specific `.css` files have CSS specific to a component. We don't have component-specific CSS for the Home component, so we don't need to make any modifications to the `home.css` file. But, we do have some global styling changes to review before we move on.

In listing 16.2, many of the UI elements have CSS classes using NativeScript themes. For example:

```
<Button class="btn btn-primary btn-rounded-sm btn-active m-r-20"  
       text="About"></Button>
```

In a NativeScript-with-Angular app, it's easy to forget that the underlying app is a NativeScript app, and that means you can still use vanilla NativeScript things like themes.

TIP To add support for themes in a NativeScript-with-Angular app, import the theme CSS file in the `app.css` file, just like you would do in a vanilla NativeScript app.

Look at the `app.css` file and you'll see the light theme already imported: `@import "nativescript-theme-core/css/core.light.css";`.

NOTE The template used to create the Pet Scrapbook Angular app, `nativescript-template-ng-tutorial`, includes the `nativescript-theme-core` npm package, so we didn't have to add the CSS import statement to the `app.css` file. But, if we hadn't used this theme, we would have needed to add the theme via npm with the following command: `npm install nativescript-theme-core --save`.

ADDING THE ANGULAR-SPECIFIC COMPONENT CODE

The final step to adding the Home component is to define the Angular-specific code in the `home.component.ts` file, by adding the contents of listing 16.3 to the file.

Listing 16.3 The `home.component.ts` file showing the Angular component declaration code

```
import { Component } from "@angular/core"; // #A  
  
@Component({ // #B  
  selector: "home", // #C  
  templateUrl: "views/home/home.html", // #D  
  styleUrls: ["views/home/home.css"] // #E  
)  
export class HomeComponent { // #F
```

```
}
```

#A Angular Component import is needed to reference @Component declaration
#B Defines the component
#C Unique identifier for the component
#D Path to the UI markup file
#E Path to the CSS styling file
#F Exports the component so Angular can use it

There's a lot of new things to talk about in listing 16.4, so we'll guide you through each one. At the top is the import statement pulling in the *Component* class definition from the `@angular/core` npm package.

TIP Every Angular component you create will require the *Component* class to be imported from the `@angular/core` package. So, get used to typing it! If you're using Visual Studio Code as an editor, there's a great extension written by Nathan Walker called NativeScript + Angular Snippets that can help you create components using snippets. Check it out at <https://marketplace.visualstudio.com/items?itemName=wwalkerrun.nativescript-ng2-snippets>.

After importing the *Component* class definition, it is used to declare the *Home* component. You'll recall this syntax (called directive syntax) from the last chapter when we looked at the `app.component.ts` file. We're using the same syntax in the *Home* component with the `@Component({ ... })` code lines.

TIP We like to think of directives like calling a function because when they're defined using the `@Component({ ... })` syntax, they can accept a variety of parameters. We are using only a subset of parameters that the `@Component` can take. A full listing of the parameters is available in the official Angular documentation at: <https://angular.io/docs/ts/latest/api/core/index/Component-decorator.html>

When creating the *Home* component, we passed three values into it: a *selector*, *templateUrl*, and *styleUrls*. The first parameter is the *selector*. The *selector* parameter is used to identify the component. We like to think of it as the identifier of a component. It also defines a custom markup tag that can be used to represent the component's contents. We'll be coming back to talk about the *selector* later when we implement page navigation, so stay tuned.

TIP Component selectors don't have to be unique within your app, but we find it easier to uniquely name them. For the purposes of this book, we'll use unique names.

The last two component directive parameters are used to define the location of the UI template (*templateUrl*) and CSS file (*styleUrls*). When they're defined, they tell Angular where it can find the files. We have named our component the *HomeComponent* and if we had any business logic, then we would build it into the *HomeComponent* class, which we will be doing a bit later in this chapter.

The final line of code (`export class HomeComponent({})`) is the Home component class exported so Angular can create this component. It's not important that you know how this is created, but just that it needs to be included in the component declaration.

TIP Per the Angular style guide, exported component class declaration should be named with the component's name followed by *Component*. For example, the Home component should be declared as `HomeComponent`. Note the capitalization of both *Home* and *Component*.

Wow, that was a lot, and sometimes, it can be tough to remember everything that needs to go into a component.

TIP If you ever get tired of typing component declarations over and over, use the power of your code editor and create shortcuts (or snippets) to make creating components easier. Also, if you're using Visual Studio Code, check out the NativeScript + Angular Snippets extension at <https://marketplace.visualstudio.com/items?itemName=wwwalkerrun.nativescript-ng2-snippets>. I know we've said it once already, but it's worth installing!

16.1.2 Loading the Home component

You'll recall from the previous chapter that the Pet Scrapbook Angular app started with a blank screen with "My App" text in the action bar (figure 16.5).

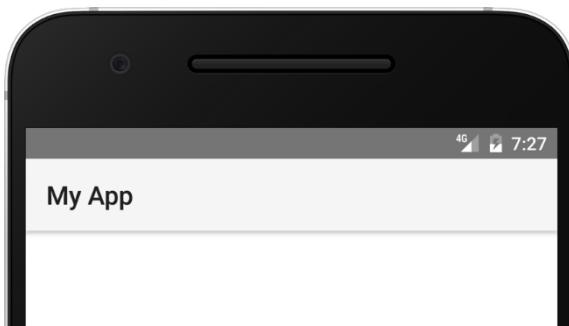


Figure 16.5 The Pet Scrapbook starts by showing a blank screen with My App text in the action bar.

Now that we have finished re-creating the *Home* page as a component, we should display it when the app launches. In a vanilla NativeScript app, we'd change the `app.js` file to load the home page, but this is Angular. As you learned in the previous chapter, the `app.module.ts` file loads the starting component of a NativeScript app via the `bootstrap` parameter of the `@NgModule` directive (listing 16.4).

Listing 16.4 The `app.module.ts` file `@NgModule` directive showing `AppComponent` as the bootstrap component

```
import { NgModule, NO_ERRORS_SCHEMA } from "@angular/core";
import { NativeScriptModule } from "nativescript-angular/nativescript.module";
```

```
import { AppComponent } from "./app.component";

@NgModule({
  declarations: [AppComponent],
  bootstrap: [AppComponent], // #A
  imports: [NativeScriptModule],
  schemas: [NO_ERRORS_SCHEMA],
})
export class AppModule {}
#A App component will be used as the first page
```

To change this to the *Home component*, we need to make minor modifications to the definition of the *App module*. Listing 16.5 shows the updated *app.module.ts* file that loads the *Home component* when the app launches.

Listing 16.5 The updated app.module.ts file to load the Home component when the app launches

```
import { NgModule, NO_ERRORS_SCHEMA } from "@angular/core";
import { NativeScriptModule } from "nativescript-angular/nativescript.module";

import { AppComponent } from "./app.component";
import { HomeComponent } from "./views/home/home.component"; // #A

@NgModule({
  declarations: [AppComponent, HomeComponent], // #B
  bootstrap: [HomeComponent], // #C
  imports: [NativeScriptModule],
  schemas: [NO_ERRORS_SCHEMA],
})
export class AppModule {}
#A Imports the Home component
#B Define the Home component as being part of this app
#C Load the Home component when the app launches
```

Listing 16.5 changes three things in the *app.module.ts* file: the *Home component* is imported, then included in the app via the `@NgModule` declarations property, and finally set as the startup component via the bootstrap property.

NOTE The declarations parameter is an array of components in the app. You can think of adding the components here as defining them in a public namespace for the entire app. The array is used by Angular, so we can use components in our app and even use components within other components.

The changes to *app.module.ts* are minor, but important. If you're just getting started with Angular, importing, declaring, and changing the bootstrap parameter may have seemed like a lot of ceremony. But, the ceremony is part of how Angular works. In fact, the pattern of importing components and adding them to the `@NgModule` directive is something you'll do often in Angular apps. Explaining the exact reasons why you need to make these changes is outside the scope of this book. Just remember that whenever you create a new component, import and declare it in *app.module.ts*. If you're interested in the details, you can read more about modules at <https://angular.io/docs/ts/latest/guide/ngmodule.html>.

TIP When you create a component, don't forget to import and declare it in the `app.module.ts` file.

WARNING Don't forget the images! Before we run the Angular app, don't forget to grab the images from the vanilla NativeScript app and place them in the images folder of the NativeScript-with-Angular version.

Now that we've made these modifications, run the app and you'll see the Home component loaded on startup (figure 16.6). It's the same view as the vanilla NativeScript app, except built with Angular!

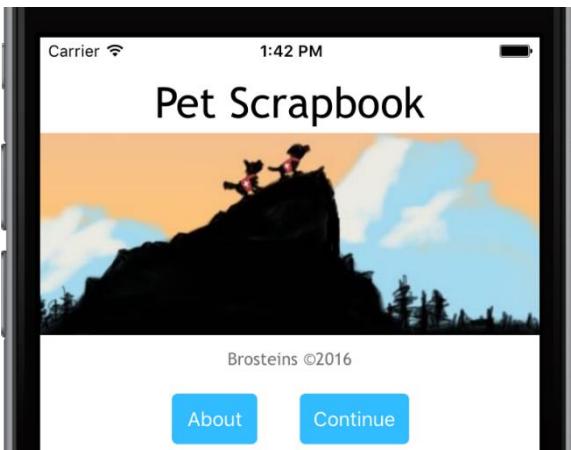


Figure 16.6 The Home page of the Pet Scrapbook Angular app written using NativeScript-with-Angular.

RECAPPING ANGULAR COMPONENTS

Nice work. Angular components can be tricky because there's a lot to remember. We'll be creating several more components throughout the rest of the book, so it's good to get familiar with them right now. Just to recap, you'll need to remember these five things about Angular components as we continue:

1. A component is the same as a vanilla NativeScript page.
2. Each component has three files: view markup (`{name}.html`), styling (`{name}.css`), and code definition (`{name}.component.ts`).
3. A component's three files should be placed in a folder within the `views` folder. The folder should be named the same as the component. For example, the `Home component` goes in the `views/home` folder.
4. The component's code definition file officially defines the component by passing in the `selector`, `templateUrl`, and `styleUrls` parameters to the `@Component` directive.
5. After creating a component, don't forget to import and declare it in the `app.module.ts` file.

16.2 Navigating between components with routing

Now that you've learned how to create your first component in a NativeScript-with-Angular app, let's create your second: the List component. The List component parallels the List page of the Pet Scrapbook, which displayed a list of the various scrapbook pages (figure 16.7).

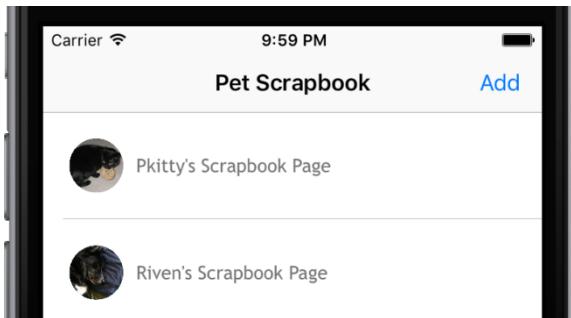


Figure 16.7 The list page of the Pet Scrapbook, which will be re-created as a List component.

After creating the component, you'll learn how to use the Angular routing system to navigate between the home and List components.

NOTE In this chapter, we won't be re-creating the entirety of the List page. Instead, we'll create a mostly-blank list page, and add to it in the next chapter.

16.2.1 Creating the List component

Let's start by creating the List component. Add a *list* folder underneath the *views* folder, then add the three List component files to the folder: *list.html*, *list.css*, and *list.component.ts*. Figure 16.8 shows the contents of the *views* folder after adding the folder and files.

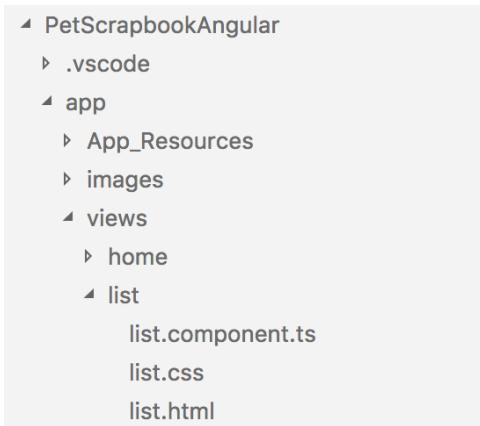


Figure 16.8 The Pet Scrapbook Angular app after adding the List component files to the project.

After adding the files, let's flesh out the component by adding some basic markup to the *list.html* file and declare the component in the *list.component.ts* file. Our component won't be using any custom CSS right now, so we'll leave it alone. The *list.html* file has a single UI element: <ActionBar title="Pet Scrapbook"></ActionBar>. Listings 16.6 shows the component declaration.

Listing 16.6 The list.component.ts file declares the List component

```
import { Component } from "@angular/core";

@Component({
  selector: "list",
  templateUrl: "views/list/list.html",
  styleUrls: ["views/list/list.css"]
})
export class ListComponent { //A
  constructor() {

  }
}
#A Note the component name follows standard style guidelines
```

There's not much to say about the List component because it's an empty shell. The final change we need to make is to tell Angular that the List component exists by updating the app module in *app.module.ts* (listing 16.7).

Listing 16.7 The app.module.ts file with the List component added

```
import { NgModule, NO_ERRORS_SCHEMA } from "@angular/core";
import { NativeScriptModule } from "nativescript-angular/nativescript.module";

import { AppComponent } from "./app.component";
import { HomeComponent } from "./views/home/home.component";
import { ListComponent } from "./views/list/list.component"; //A

@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    ListComponent // #A
  ],
  bootstrap: [HomeComponent],
  imports: [NativeScriptModule],
  schemas: [NO_ERRORS_SCHEMA],
})
export class AppModule {}
#A Import the List component and add it to the module declarations
```

As we did earlier in this chapter when we added the Home component, the List component needs to be imported and declared in the app module.

We'll continue to build up the List component in the next chapter, but before we do that, you need to learn about navigating between pages in NativeScript-with-Angular apps.

16.2.2 Page navigation in NativeScript-with-Angular

In the last chapter, we briefly described how NativeScript-with-Angular apps navigate between pages with the Angular *router*.

DEFINITION The router allows you to navigate from one page to another in a NativeScript-with-Angular app.

The concept of a router and routing between pages is borrowed from the web, where various pages of a web app have a unique URL, for example, <http://foo.com/foo> and <http://foo.com/bar>. This allows users to either enter a URL, click a link on a page, or use the forward and back buttons to navigate between pages with unique addresses.

But, you may be wondering how this relates to NativeScript-with-Angular apps. The official Angular documentation on routing and navigation sums it up nicely by stating that Angular borrows from the navigation concepts of the web. The Angular router "...can interpret a browser URL [or URL fragment] as an instruction to navigate to a client-generated view...you can bind the router to links on a page and it will navigate to the appropriate application view when the user clicks a link. And the router logs activity in the browser's history journal so the back and forward buttons work as well" (Angular documentation: <https://angular.io/docs/ts/latest/guide/router.html>).

NOTE Providing an abstraction over navigation concepts allows Angular to be multi-platform and not be locked into the web. In HTML apps, the router interacts with the browser's URL and HTTP requests for pages, but in NativeScript-with-Angular apps, there is no browser or URL. Instead, the router interacts with the native platform's history.

It's not important for you to know much more than this to use Angular routes, so we're not diving deeper into how the router works and how it interacts with your app. If you're interested in the finer details, check out the official documentation at <https://angular.io/docs/ts/latest/guide/router.html>.

DEFINING APP ROUTES

To start using the router in the Pet Scrapbook Angular app, there are a few steps we'll need to take:

- Step 1: Define the routable components
- Step 2: Configure the app module to use routing
- Step 3: Create a router outlet in the app component to manage control navigation
- Step 4: Change the startup page to the app component
- Step 5: Add routing logic to the Home component to navigate to the List component

It's ok if you don't understand completely the steps we just described. We'll address each one as we proceed.

STEP 1: DEFINE THE ROUTABLE COMPONENTS

The first step is to define the components within our app that will be considered routable.

DEFINITION Routable components are components that can be navigation destinations in a NativeScript-with-Angular app. Up until now, you've seen only components that act as entire pages, but it's possible to create components that are used as subcomponents inside another component. Often, subcomponents will never be displayed alone, so there's no need to navigate to them directly.

Let's start by creating a new file named `app.routing.ts` in the app folder. This file is a special file used by Angular to define the various components we can navigate (or route) to.

TIP You can define multiple routes that navigate to the same component. For example, listing 16.8 assigns the default route and home route to the Home component.

We only have two components, and both are considered routable: `Home` and `List`. Add these components to the `app.routing.ts` file, as shown in listing 16.8.

Listing 16.8 The `app.routing.ts` file defines the routes and routable components of the Pet Scrapbook app

```
import { HomeComponent } from "./views/home/home.component"; // #A
import {ListComponent} from "./views/list/list.component"; // #A

export const routes: any = [ // #B
  { path: "", component: HomeComponent }, // #B
  { path: "home", component: HomeComponent }, // #B
  { path: "list", component: ListComponent } // #B
];

export const navigatableComponents: any = [ // #C
  HomeComponent, // #C
  ListComponent // #C
];
#A Import routable components
#B Define Angular routes
#C Create a list of components that we want to navigate to
```

There are three things going on in the routing file. First, we import each of the components we'd like to route to, so we can use them throughout this file. Second, we define an array of routes, which is a set of key value pairs with a *path* and a *component*. The path is considered the route, or short-hand reference, we can use to refer to the component (which has a longer name).

NOTE The routes array contains a special route with an empty path. This is called the default route, and will be used later as the route that gets loaded by default when our app starts up.

Last, a separate array of components is established to track the components to which we want to navigate. You'll notice that these are the same components defined in the routes array, but each component is listed only once.

STEP 2: CONFIGURE THE APP MODULE TO USE ROUTING

The second change will be made to the `app.module.ts` file. We'll import the NativeScript-specific routing module and configure it to use the routes we defined in the previous step. Listing 16.9 shows the updates.

Listing 16.9 The app.module.ts updated with the routing configuration

```
import { NgModule, NO_ERRORS_SCHEMA } from "@angular/core";
import { NativeScriptModule } from "nativescript-angular/nativescript.module";
import { NativeScriptRouterModule } from "nativescript-angular/router"; // #A
import { routes, navigatableComponents } from "./app.routing";           // #A

import { AppComponent } from "./app.component";
import { HomeComponent } from "./views/home/home.component";
import { ListComponent } from "./views/list/list.component";

@NgModule({
  declarations: [
    AppComponent,
    ...navigatableComponents // #B
  ],
  bootstrap: [
    HomeComponent
  ],
  imports: [
    NativeScriptModule,
    NativeScriptRouterModule,                                // #A
    NativeScriptRouterModule.forRoot(routes) // #A
  ],
  schemas: [NO_ERRORS_SCHEMA],
})

export class AppModule {
}

#A Import the components needed for routing
#B TypeScript short hand for concatenating navigatableComponents array to the declarations array
```

At the top of listing 16.9, the *NativeScript router module* is imported with the arrays we created in the previous step.

DEFINITION The NativeScript router module is a special version of the Angular router, written specifically for NativeScript apps. Like the Angular router, it's responsible for navigating between components and interacting with the underlying native app systems (Android and iOS).

After adding the router module, it needs to be imported into the module declaration by adding it to the `imports` property of `@NgModule`. In addition to importing `NativeScriptRouterModule`, the `NativeScriptRouterModule.forRoot(routes)` command needs to be imported.

NOTE The `.forRoot(routes)` command is important because it configures and loads the routes from the routes array. Without this command, Angular wouldn't know about any of the valid routes (even though we imported the routing module).

The components added to the `navigatableComponents` array also need to be added to the declarations array of the module. You'll notice we used three dots before the array name: `...navigatableComponents`. This isn't a typo, but instead a TypeScript shortcut for adding the contents of the `navigatableComponents` array to the `declarations` array.

NOTE When we added the `navigatableComponents` array, we removed the `HomeComponent` and `ListComponent` references from the declarations array. That's because they were added to the `navigatableComponents` array.

STEP 3: CREATE A ROUTER OUTLET IN THE APP COMPONENT TO MANAGE CONTROL NAVIGATION

Now that we've registered our routes with our app module, let's step back to understand how our app will load new components as we route to them. Grasping how routing works can be a bit confusing, so we'll look at the code first, then explain what's happened.

Let's start by updating the App component. We'll make a small change to the `app.component.ts` file, and in the next step change our app to start up with the app component.

NOTE Yes, you read that correctly. We're revisiting the App component. You may recall that we used the App component briefly in the previous chapter, but then changed the startup component to the Home component. We're going to be reusing the App component to assist with routing. We'll explain.

Listing 16.10 shows the changes you need to make to the `app.component.ts` file to define the router outlet.

Listing 16.10 The `app.component.ts` file that defines the router outlet for the Pet Scrapbook

```
import { Component } from "@angular/core";

@Component({
  selector: "my-app",
  template: "<page-router-outlet></page-router-outlet>" // #A
})
export class AppComponent {}  
#A Change the template to load Angular's page router outlet component
```

When we introduce routing into an Angular app, the `page router outlet` is added into the startup component's UI markup.

DEFINITION The page router outlet is an Angular component that knows how to dynamically load other components when you navigate to them.

This may seem confusing, but remember that Angular is a single-page application (SPA) framework. In NativeScript-with-Angular apps, the app module dynamically loads new components and places them into the UI markup area defined by the page router outlet. Figure 16.9 shows how this works.

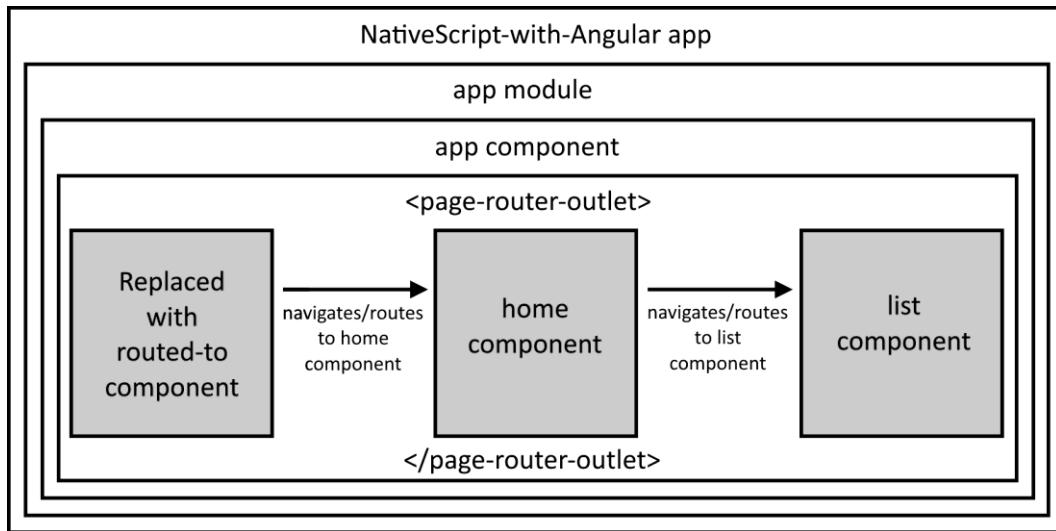


Figure 16.9 The page router outlet component dynamically loads other components into it when navigated/routed-to.

There's a lot going on in figure 16.9, so let's break it down. On the far left is a visualization of a NativeScript-with-Angular app. The outer-most layer is the app itself, running NativeScript-with-Angular. When the app loads, the *app module* is loaded, making the second layer. The *app module* then loads the *app component*, which contains the `<page-router-outlet></page-router-outlet>` component. When the Home component is navigated-to (or routed-to), the content of the page router outlet is replaced with the Home component's UI markup. Then, when the List component is routed-to, its UI markup is dynamically loaded into the page router outlet.

NOTE You'll recall that components don't need to include the `page` element in the UI. This is possible because the `<page-router-outlet></page-router-outlet>` markup emits the `page` element and because of the dynamic SPA nature of Angular.

Now we've covered how the page router outlet dynamically loads components, let's return to the app component and configure our app to use it as the startup component.

STEP 4: CHANGE THE STARTUP PAGE TO A NEW COMPONENT NAMED APP

In the last chapter, you learned how to change the startup component by changing the value of the bootstrap component. Set the startup component to the app component, as shown in listing 16.11.

Listing 16.11 The app.module.ts updated to load the app component as the startup component

```
import { NgModule, NO_ERRORS_SCHEMA } from "@angular/core";
import { NativeScriptModule } from "nativescript-angular/nativescript.module";
import { NativeScriptRouterModule } from "nativescript-angular/router";
import { routes, navigatableComponents } from "./app.routing";
```

```
import { AppComponent } from "./app.component";
import { HomeComponent } from "./views/home/home.component";
import { ListComponent } from "./views/list/list.component";

@NgModule({
  declarations: [
    AppComponent,
    ...navigatableComponents
  ],
  bootstrap: [
    AppComponent //#A
  ],
  imports: [
    NativeScriptModule,
    NativeScriptRouterModule,
    NativeScriptRouterModule.forRoot(routes)
  ],
  schemas: [NO_ERRORS_SCHEMA],
})
export class AppModule {
}

#A The bootstrap property controls the startup component
```

After changing the startup component, our app will load the `<page-router-outlet></page-router-outlet>` UI markup when the app loads. Let's run the Pet Scrapbook and see what it looks like (figure 16-10).

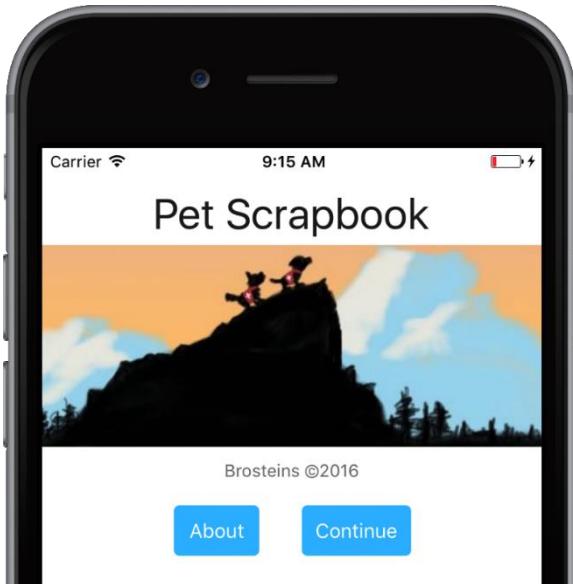


Figure 16.10 The Pet Scrapbook after changing the startup component to the app component.

That's funny. We just changed the startup component to be the app component, right? But, the Home component is being displayed. What's happened?

Back in step 2, you'll recall we added the `NativeScriptRouterModule.forRoot(routes)` command to the app module declaration. This command loaded our configured routes (the `routes` array) into the router. The `routes` array had three values, each defining a different path and component for the Pet Scrapbook (listing 16.12).

Listing 16.12 The routes array, establishing three routes for the Pet Scrapbook

```
export const routes: any = [
  { path: "", component: HomeComponent }, //#A
  { path: "home", component: HomeComponent },
  { path: "list", component:ListComponent }
];
#A The default route
```

Previously, we discussed the home and list routes, but we didn't cover the route with an empty string for the path: `{ path: "", component: HomeComponent }`. This is called the *default route*.

DEFINITION The default route is the route that Angular navigates to when the app loads when the app loads.

Now that you know how the default route works, it's clear why the Home component was displayed when the Pet Scrapbook launched.

STEP 5: ADD ROUTING LOGIC TO THE HOME COMPONENT TO NAVIGATE TO THE LIST COMPONENT

The final step is to add routing logic into the Home component so we can navigate from the Home component to the List component. In a vanilla NativeScript app, we would add a *tap event* to the UI markup of the home page, then add a tap-event handler to the JavaScript. NativeScript-with-Angular apps use the same process, but with slightly different syntax. We'll cover the syntax difference in the next chapter when we cover data binding. Right now, you need to know that by adding the `(tap)` attribute, Angular knows to call the `onContinueTap()` function when the button is tapped.

WARNING We'll be using Angular data binding in the next several listings, even though you haven't learned about it yet. For now, just follow along. We'll cover data binding in the next chapter with greater detail.

Let's start by adding the tap event to the Home component's UI by updating `home.html`. Change the Continue button's UI markup by adding the `(tap)="onContinueTap()"` attribute with the following code:

```
<Button class="btn btn-primary btn-rounded-sm btn-active m-1-20"
  text="Continue" (tap)="onContinueTap()"></Button>
```

Next, let's define the `onContinueTap()` function by updating the `home.component.ts` file. Listing 16.13 shows the changes.

Listing 16.13 The home.component.ts file updated to navigate to the List component when the Continue button is tapped

```
import { Component } from "@angular/core";
import { RouterExtensions } from "nativescript-angular/router"; // #A

@Component({
  selector: "home",
  templateUrl: "views/home/home.html",
  styleUrls: ["views/home/home.css"]
})
export class HomeComponent {
  constructor(private routerExtensions: RouterExtensions) { // #B
  }

  onContinueTap(): void {
    this.routerExtensions.navigate(["list"]); // #C
  }
}

#A Import the NativeScript-Angular router
#B Load the Angular router into the HomeComponent class
#C Handle the tap event and navigate to the List component using the router
```

At the top of the Home component, we import the Angular router, then add a router reference to the constructor of the Home component. After adding the references, the tap-event handler is defined so it navigates to the list route.

NOTE The `navigate()` function is part of the `nativescript-angular` module and takes an array of values. The first value is the name of the route. Additional values can be supplied, but we're not going to cover those values until the next chapter. If you'd like to learn more now, check out <https://docs.nativescript.org/core-concepts/angular-navigation>.

That should do it. Run the Pet Scrapbook Angular app and tap the *Continue* button, which will navigate from the Home component to the List component, as shown in figure 16.11.

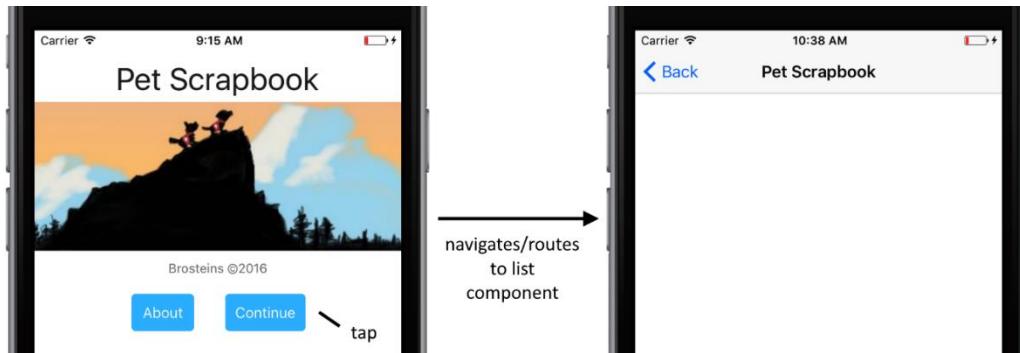


Figure 16.11 When the Continue button is tapped, the Pet Scrapbook routes to the List component.

Great work! You've learned the basics of Angular components and routing, while re-creating several key pages of the Pet Scrapbook: home and list. In the next chapter, you'll learn about data binding, finish the List component, and create the details component.

16.3 Summary

In this chapter, you learned:

- Angular components have three pieces: a view (written in UI markup), styling (CSS), and code (TypeScript)
- Components must be declared in the app module
- The `<page-router-outlet></page-router-outlet>` loads the default route when the app starts, as defined in the app module by calling `NativeScriptRouterModule.forRoot(routes)`.

16.4 Exercise

Using your knowledge of NativeScript and Angular learned in this chapter, add a new component named *about*, and navigate to the *about component* when the Home component *About* button is tapped.

16.5 Solutions

There are seven steps to creating an *about* component and navigating to it when the *About* button is tapped on the *Home* component:

- Step 1: Create the *about* component files
- Step 2: Add UI markup to the *about* component
- Step 3: Define the *about* component code
- Step 4: Import the *about* component into the app module
- Step 5: Add the *about* component route to the app module
- Step 6: In the *Home* component's UI markup, add a tap event to the *About* button
- Step 7: Handle the *About* button's tap event, navigating to the *about route*

16.5.1 Step 1: Create the *about* component files

Create a folder named *about* in the *app/views* folder. Add three files to the *about* folder: *about.html*, *about.css*, and *about.component.ts*.

16.5.2 Step 2: Add UI markup to the *about* component

Add the following to the *about.html* file. We didn't define the UI for the *about* page in the vanilla NativeScript app, so feel free to put in any content you wish. We added a label, just so you know the navigation worked.

```
<StackLayout>
    <Label text="About the Pet Scrapbook"></Label>
</StackLayout>
```

16.5.3 Step 3: Define the about component code

Add the contents of listing 16.14 to the `about.component.ts` file.

Listing 16.14 The `about.html` file, containing the UI markup for the about component

```
import { Component } from "@angular/core";

@Component({
  selector: "about",
  templateUrl: "views/about/about.html",
  styleUrls: ["views/about/about.css"]
})

export class AboutComponent { }
```

16.5.4 Step 4: Import the about component into the app module

Update the `app.module.ts` file to include the about component, as shown in listing 16.15.

Listing 16.15 The `app.module.ts` updated to include the about component

```
import { NgModule, NO_ERRORS_SCHEMA } from "@angular/core";
import { NativeScriptModule } from "nativescript-angular/nativescript.module";
import { NativeScriptRouterModule } from "nativescript-angular/router";
import { routes, navigatableComponents } from "./app.routing";

import { AppComponent } from "./app.component";
import { HomeComponent } from "./views/home/home.component";
import {ListComponent} from "./views/list/list.component";
import {AboutComponent} from "./views/about/about.component";

@NgModule({
  declarations: [
    AppComponent,
    ...navigatableComponents
  ],
  bootstrap: [
    AppComponent
  ],
  imports: [
    NativeScriptModule,
    NativeScriptRouterModule,
    NativeScriptRouterModule.forRoot(routes)
  ],
  schemas: [NO_ERRORS_SCHEMA],
})
export class AppModule { }
```

16.5.5 Step 5: Add the about component route to the app module

Update the `app.routing.ts` file to include a route for the about component, as shown in listing 16.16.

Listing 16.16 The `app.routing.ts` file updated to include a route for the about component

```
import { HomeComponent } from "./views/home/home.component";
```

```
import {ListComponent} from "./views/list/list.component";
import {AboutComponent} from "./views/about/about.component";

export const routes: any = [
  { path: "", component: HomeComponent },
  { path: "home", component: HomeComponent },
  { path: "list", component: ListComponent },
  { path: "about", component: AboutComponent }
];

export const navigatableComponents: any = [
  HomeComponent,
  ListComponent,
  AboutComponent
];
```

16.5.6 Step 6: In the Home component's UI markup, add a tap event to the About button

Add a tap-event handler to the About button on the Home component (*home.html*):

```
<Button class="btn btn-primary btn-rounded-sm btn-active m-r-20"
  text="About" (tap)="onAboutTap()"></Button>
```

16.5.7 Step 7: Handle the About button's tap event, navigating to the about route

Add the tap-event handler to the Home component code file, as shown in listing 16.17.

Listing 16.17 The *home.component.ts* file updated to navigate to the about component when the About button is tapped

```
import { Component } from "@angular/core";
import { RouterExtensions } from "nativescript-angular/router";

@Component({
  selector: "home",
  templateUrl: "views/home/home.html",
  styleUrls: ["views/home/home.css"]
})
export class HomeComponent {
  constructor(private routerExtensions: RouterExtensions) {}

  onContinueTap(): void {
    this.routerExtensions.navigate(["list"]);
  }

  onAboutTap(): void {
    this.routerExtensions.navigate(["about"]);
  }
}
```

17

Angular databinding and services

This chapter covers

- Implementing databinding in Angular
- Services in Angular
- Navigating using modal dialogs

In the previous chapter, you continued building the NativeScript-with-Angular version of the Pet Scrapbook by adding the Home and List components. You also learned how Angular routing works and used the `<page-router-outlet></page-router-outlet>` to dynamically load the List component.

There's a lot to cover in this chapter, but we're going to approach it just like we did with the original Pet Scrapbook app. We'll start by introducing you to how Angular databinding works, teaching you one-way binding, event binding, and then two-way binding. Then, you'll learn how services are created and used in an Angular app. Finally, we'll finish the chapter by teaching you how to open components as modal dialogs.

Let's get started!

17.1 Databinding with Angular

In chapter 8, we described databinding as the process of linking together JavaScript objects and UI elements. In a vanilla NativeScript app, the JavaScript object we were binding to was an observable, loaded from the *data/observable* NativeScript core module. In the UI layer, UI element attributes were linked with NativeScript's mustache syntax. For example, to bind the *name* property of an observable to the *text* attribute of a label element you would use: `<Label text="{{ name }}" />`.

The same concepts of binding a code object to UI elements still apply in NativeScript-with-Angular apps. In general, the differences come down to databinding syntax in the UI and the type of JavaScript object you bind to. We're not going to go into the details of the differences right now, but we'll explore them as you learn how Angular databinding works.

Let's start our exploration by revisiting the List component.

17.1.1 Adding one-way databinding to the List component

When we left the List component in the last chapter, it was shell only, created with the purpose of having a navigation destination to use while you learned about routing. As a reminder, figure 17.1 shows what we're starting with.

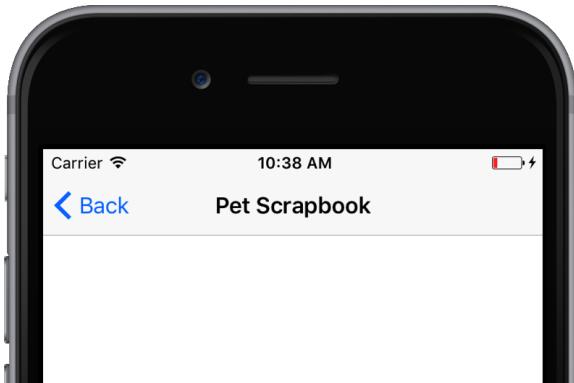


Figure 17.1 At the end of chapter 16, the List component UI markup contains only an action bar at the top.

MODIFYING THE LIST COMPONENT CODE LAYER

Let's start from the List component's code layer first, defining the object that we will bind to in the UI layer. You'll recall that the list page was bound to an observable array in the vanilla NativeScript app. In NativeScript-with-Angular apps, it's a bit easier because we can use a simple TypeScript array as the databinding source. Listing 17.1 shows the updated *list.component.ts* file, using an array to store scrapbook pages.

Listing 17.1 *list.component.ts* file with an array added for the databinding source

```
import { Component } from "@angular/core";
import { Page } from "../../models/page"; // #A

@Component({
  selector: "list",
  templateUrl: "views/list/list.html",
  styleUrls: ["views/list/list.css"]
})
export class ListComponent {
  pages: Array<Page>; // #B

  constructor() { }
}

#A Import a class to represent individual scrapbook pages
#B Create an array of scrapbook pages
```

It almost feels like we're cheating because the code we've added is so minimal. Inside of the *ListComponent* class, we declared an array to hold individual scrapbook pages. To make it easier (and to

use an object-oriented feature of TypeScript) we created a new class named *Page*. You'll notice that the *Page* class was imported at the top of the component. Before we go any further, create the *Page* class. Add a folder named *models* to the *app* folder, then add a file named *page.ts* to the *models* folder. Place the code from listing 17.2 into the *page.ts* file.

Listing 17.2 The page.ts file defines a re-useable Page class that represents a single scrapbook page

```
import { ImageSource } from "image-source"; // #A

export class Page {
    Id: number;
    Title: string;
    Age: string;
    BirthDate: any;
    Gender: string;
    Lat: number;
    Long: number;
    Image: ImageSource;
    ImageBase64: string;
}
#A The image source references the Pet's photo
```

We haven't talked about class definitions in TypeScript before, and we're not going to dive deep, because we're assuming you're familiar with object-oriented programming. We will, however, point out some TypeScript syntax. The *Page* class has defined eight public properties. It may look strange because there's no *var* or *let* keyword—just the property name followed by its declared data type.

TIP To declare a public class property in TypeScript, use the form `{propertyName}: {datatype};` within the class declaration. For example, `Id: number;` declares a public property named *Id* of type *number*.

NOTE The *Page* class is also a good example of writing maintainable code. With the definition of the *Page* class in a single file, we can reuse it throughout our app. If any property needs to change in the future, it can be changed in a single place.

Now that you know about the *Page* class and that we'll be using it throughout the app, let's get back to the array declaration in the *list.component.ts* file. You'll recall we added a public array property named *pages* of type *Page*: `pages: Array<Page>;`. First, this array will be the object the UI markup will bind to. Second, notice that there's no mention of an *observable* or *observable array*.

NOTE In NativeScript-with-Angular apps, you don't need to use the *data/observable* or *data/observable-array* core module because Angular is smart. When you bind to a public property of a class (like the *pages* property), observable-like behavior gets built into the property automatically. And the best part is you don't need to do anything special. Cool, right?

UPDATING THE LIST COMPONENT UI

Next, let's turn our attention to the UI markup of the List component. To re-create the original Pet Scrapbook list page, add the following UI markup to the list.html file (listing 17.3).

Listing 17.3 The updated list.html file incorporating a list view

```
<ActionBar title="Pet Scrapbook">
</ActionBar>

<GridLayout rows="*" columns="*">
  <ListView class="list-group" [items]="pages"> // #A
    <template let-item="item"> // B
      <StackLayout orientation="horizontal" class="list-group-item">
        <Image class="thumb img-circle" [src]="item.Image"></Image> // C
        <Label class="list-group-item-text" [text]="(item.Title === null ? 'New' : item.Title + '\'s') + ' Scrapbook Page'"></Label> // C
      </StackLayout>
    </template>
  </ListView>
</GridLayout>
```

#A items collection is data-bound to the pages property
#B templates allow you to iterate over a collection with let-item property
#C template contents are rendered for each item in the items collection

Most of the *list.html* file is familiar. But, the Angular databinding syntax is a little different. The first difference we'll point out is the list view items attribute. It's data-bound by placing square brackets around the attribute's name: `[items]="pages"`, meaning that the *items* attribute is bound to the *pages* property of the List component.

TIP To bind a UI element attribute to a component property, wrap the attribute in square brackets `[]` and set its value to the name of the component property.

It's not important that you understand why Angular works this way. Instead, just remember that it's the same as using vanilla NativeScript databinding like `items="{{ pages }}"`.

NOTE The `[attribute]="property"` syntax is specifically known as one-way databinding. In chapter 8, you learned about one-way NativeScript databinding, where changes made to the JavaScript object are reflected in the UI, but not the reverse. Angular one-way databinding works the same way. If you're wondering about two-way databinding, hold on because we'll cover it later in this chapter.

Another difference in the List component's UI markup is how item templates are rendered: `<template let-item="item">...</template>`. In vanilla NativeScript, there's a special UI element named `ListView.itemTemplate` that is used to define the repeating item template. But, NativeScript-with-Angular apps can take a simplified approach by using the Angular syntax feature *let-item*.

DEFINITION Angular's let-item is a special syntax that acts as a for-loop iterator. When used in conjunction with a template element as the child element of a data-bound array (like our `[items]="pages"` bound list view), the contents of the template are repeated for each item in the array.

The let-item syntax can be a bit confusing the first time you see it, but we like to think of it like a for loop inside of the UI markup. It creates a reference to each item in the list view, using the value of the let-item attribute as a variable that can be used in other databinding expressions within the template.

This leads us to the last change in the component, which is databinding the image and label within the template:

```
<Image class="thumb img-circle" [src]="item.Image"></Image>
<Label class="list-group-item-text"
  [text]="'(item.Title === null ? 'New' :
  item.Title + '\'s') + ' Scrapbook Page'"></Label>
```

Inside of the template, the `let-item="item"` syntax creates a local variable named `item`. The variable references the current iterator instance of the `Page` class. Because it's an instance of the `Page` class, we have access to its properties, which can be seen in the image's `src` attribute: `[src]="item.Image"`.

Before we move on, we want to point out that the same databinding expression syntax (like the syntax used in the label's `text` attribute) works in both vanilla NativeScript and NativeScript-with-Angular apps.

NOTE You've already learned how databinding expressions work in vanilla NativeScript, so we're not going into detail here. If you'd like a refresher, check out chapter 9.

CHECKING OUT THE CHANGES

Nice work! The List component now keeps track of an array of Pages, which is bound to the list view in the UI markup. As scrapbook pages are added and updated in the `pages` array, the component's UI will stay in sync.

Let's look at the results. Run the Pet Scrapbook Angular app and navigate to the List component by tapping the Continue button. The List component now has an empty list view displayed, as shown in figure 17.2.

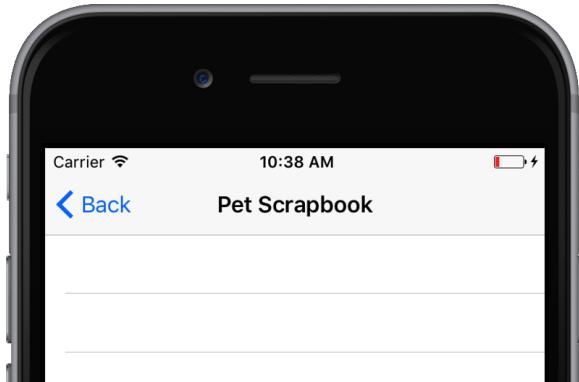


Figure 17.2 The List component UI markup updated to include the data-bound list view.

That's it. You've learned the basics of databinding in a NativeScript-with-Angular app. And, it feels a bit easier than vanilla NativeScript: add a public property to the component, then update the UI markup to use Angular databinding syntax.

But, this milestone feels bittersweet because we've bound the List component to an empty array of scrapbook pages. In the next section, we'll fix this by recreating the file system service to load and save scrapbook pages to the native device.

17.2 Creating and using services

In chapter 9, you learned how to use the file-system core module to read and write data to the native device's file system. When we used the file-system core module, we wrapped it in a special JavaScript function we called a *service class*.

DEFINITION A service class is a collection of reusable code that can be shared throughout an application to perform a specific group of related functionality. Service classes typically create an internal API or intermediate layer of functionality in your code and sit between the front-end UI layer of your application and data or file-system access layers. Service classes generally contain business logic.

Good news. The concept of services (like the file-system service we created in the Pet Scrapbook app) is the same in NativeScript with Angular apps. In fact, the concept of services in Angular apps are considered a good practice.

TIP In NativeScript-with-Angular apps, you should wrap the functionality in a service when you need to interact with remote data sources, the file system, or a hardware component that may be slow to respond. Other reasons you may consider using a service are to hide the implementation of business logic and promote code reuse.

Now that we've reinforced the concept of a service, let's create a re-useable service to read and write scrapbook page data from the file system.

17.2.1 Page service

In the Pet Scrapbook Angular app, the functionality of a file-system service is largely unchanged. But, we need to make several changes.

First, the service formerly known as the file system service needs a better name: let's call it the page service. The name file-system service wasn't a bad name, but it was too specific and referenced the implementation used to access data (via the file-system core module) instead of describing the type of data (scrapbook pages). Seeing that the name of this service is incorrect may feel like a stretch now, but imagine you need to retrieve various types of data from the file system: photos, contacts, locations, and so on. Now imagine that even though all this data lives on the file system, it is accessed from different locations and in slightly different ways. Calling our service file system service now seems silly: how are other developers supposed to know the file-system service can access all this data? Wouldn't it be better to have multiple services, each named to describe the type of data they're responsible for accessing? Do you think photo service, contact service, and location service are more descriptive names? We do.

TIP Services should be named to describe the type of data they retrieve, not the methodology used to retrieve the data.

The second change we'll be making to the original file-system service is rewriting it in TypeScript using a class. You should be familiar with TypeScript classes by now, but if you need a reminder, check out the official TypeScript documentation at <https://www.typescriptlang.org/docs/handbook/classes.html>.

CREATING THE PAGE SERVICE

Before we jump into the code, create a `services` folder inside of the `app` folder, then add a file named `page.service.ts` to the services folder.

TIP The Angular style guide is clear on where service classes should be created (in the `services` folder) and how they should be named. As you create services in your own project, be sure to name them using the following convention: `{name}.service.ts`. For more style guide tips related to services, check out <https://angular.io/styleguide#!#services>.

After creating the `page.services.ts` file, add the contents of listing 17.4 to it.

Listing 17.4 The `page.service.ts` file defining the page service, which reads and writes scrapbook page data from the file system

```
import { Injectable } from "@angular/core"; // #A
import { Page } from "../models/page";
import * as fileSystem from "file-system";
import * as image from "image-source";

@Injectable() // #B
export class PageService {
  getPage(id: number): Page {
    let pages = this.getPages();
    let index = this.findPageIndex(pages, id);

    if (index === -1)
```

```
        return null;

    return pages[index];
}

getPages(): Array<Page> {
    let file = fileSystem.knownFolders.documents().getFile("scrapbook.json");
    let pages = file.readTextSync().length === 0
        ? new Array<Page>()
        : <Array<Page>>JSON.parse(file.readTextSync());

    pages.forEach((page) => {
        page.Image = image.fromBase64(page.ImageBase64);
    });

    return pages;
}

savePage(scrapbookPage: Page): void {
    let file = fileSystem.knownFolders.documents().getFile("scrapbook.json");
    let pages = this.getPages();
    let index = this.findIndex(pages, scrapbookPage.Id);
    let page = new Page();

    page.Id = scrapbookPage.Id;
    page.Title = scrapbookPage.Title;
    page.Gender = scrapbookPage.Gender;
    page.Age = scrapbookPage.Age;
    page.BirthDate = scrapbookPage.BirthDate;
    page.ImageBase64 = scrapbookPage.ImageBase64;
    page.Lat = scrapbookPage.Lat;
    page.Long = scrapbookPage.Long;

    if (index !== -1) {
        pages[index] = scrapbookPage;
    }
    else {
        pages.push(scrapbookPage);
    }

    var json = JSON.stringify(pages);
    file.writeText(json);
}

private findPageIndex(pages: any, id: number): number {
    return pages.findIndex(function (element) {
        return element.Id === id;
    });
}
}

#A Import the Injectable decorator
#B Allow the page service to be injected as a dependency into other classes
```

There's a lot of things you'll recognize from the original file-system service inside of the page service. Specifically, the `getPages()` and `savePage()` functions and their use of the `file-system` core module. There

are also several new items to discuss, including the `getPage()` function. We won't be using this function immediately, but it will come in handy as we re-create the details page.

A second change is the name of the service class: `PageService`. Following the Angular style guide and our recommendations on naming services, the class name reflects its file name of the data type (`Page`) followed by `Service`.

The final change is the addition of the `@Injectable` decorator above the `PageService` class. You learned about decorators in the last chapter when we discussed the `@Component` and `@NgModule` decorators. The `@Injectable` decorator is a special decorator that tells other classes that the `PageService` class can be injected as a dependency into other classes automatically. You learned about dependency injection briefly in chapter 15, but you can read more about it and how it works in Angular apps at <https://angular.io/docs/ts/latest/guide/dependency-injection.html>.

UPDATING THE LIST COMPONENT TO USE THE PAGE SERVICE

Now that we've created a re-useable service, let's return to the List component and use the service to load the scrapbook pages when the component loads. Update the List component's `list.component.ts` file to use the page service as shown in listing 17.5.

Listing 17.5 The `list.component.ts` file updated to inject the page service and load scrapbook pages into the page array

```
import { Component, OnInit } from "@angular/core"; // #A
import { Page } from "../../models/page";
import { PageService } from "../../services/page.service" // #B

@Component({
  selector: "list",
  providers: [ PageService ], // #B
  templateUrl: "views/list/list.html",
  styleUrls: ["views/list/list.css"]
})

export class ListComponent implements OnInit {
  pages: Array<Page>;

  constructor(private pageService: PageService) { } // #C

  ngOnInit(): void { // #D
    this.pages = this.pageService.getPages(); // #D
  } // #D
}

#A import the OnInit interface to tap into the ngOnInit lifecycle hook
#B the providers array tells Angular to create an instance of the PageService when a ListComponent is created
#C inject the PageService created in the providers array into the class
#D tap into the ngOnInit lifecycle event to get the list of pages
```

At first, you might feel overwhelmed by the code we added to the List component, so let's break it down into two pieces: injecting the page service and using it to get data.

INJECTING THE PAGE SERVICE

In NativeScript-with-Angular apps, you use dependency injection to get an instance of service classes within another class. You'll remember the `@Injectable` decorator from earlier in this chapter: that allows the page service to be injected into other classes. Dependency injection in Angular is something that happens automatically, but you need to do make sure it's configured properly by adding the `providers` property to the `@Component` decorator.

DEFINITION The `providers` property is an array of classes that tells Angular what to create an instance of when a component is created. For example, `providers: [PageService]` tells Angular to create an instance of the `PageService` class when a `List` component is created.

After declaring the page service as provider for the `List` component, all that's left is to add the page service as a constructor parameter: `constructor(private pageService: PageService)`. After adding this parameter, Angular automatically creates a private class variable named `pageService` that can be accessed from anywhere in the class by using `this.pageService`.

USING THE PAGE SERVICE

After injecting the page service into the `List` component, it's easy to get a list of pages: `this.pageService.getPages();`. But, you may want to put this service call into the constructor. Don't do it. Resist.

TIP Per the Angular documentation, constructors are no place for complex logic, or loading data from a service. Constructors are for simple object initialization and writing up properties. Complex logic operations (like service calls) that need to occur when a component is created should be done in the `ngOnInit` lifecycle hook.

DEFINITION *Lifecycle hooks* are a way of tapping into the key life moments of Angular components, allowing you the execute code when an event happens. There are a variety of lifecycle hooks to tap into. One event, `ngOnInit` is called when a component is initialized and displays data-bound properties. If you're using services to get data to display in the UI, retrieving that data during the `ngOnInit` lifecycle hook is the right time to do it. To learn about other lifecycle hooks, check out <https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html>.

We tapped into the `ngOnInit` lifecycle hook in the `List` component and used it to load scrapbook pages from the page service in listing 17.5. To do this, we had to do three things:

3. Import the `OnInit` interface from the `@angular/core` module: `import { Component, OnInit } from "@angular/core";`.
4. Add the `OnInit` interface to the `List` component class declaration: `class ListComponent implements OnInit { ... }.`
5. Implement the `OnInit` interface by adding the `ngOnInit() : void { ... }` function to the `List` component class.

After implementing the *OnInit* interface, we loaded scrapbook pages from the file system using the page service: `this.pages = this.pageService.getPages();`.

WARNING It's easy to get confused when tapping into lifecycle hook like *ngOnInit*. The interface you need to implement is named *OnInit*, but the function that needs implemented in the *OnInit* interface is named *ngOnInit*. Be careful, and mind your capitalization. If you can't remember, don't hesitate to reference <https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html>.

Unfortunately, there's not much to see when we run the Pet Scrapbook Angular app again: that's because loading scrapbook data from an empty file still shows the same empty list of pages. But, running the app right now can help you verify that everything is working as expected. If all is well, you should see something similar for figure 17.3.

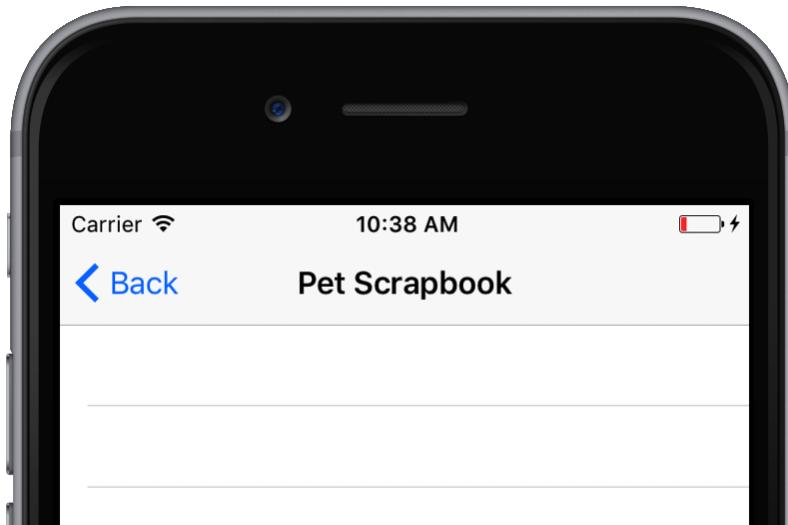


Figure 17.3 The List component after injecting the page service to load scrapbook pages from the file system. You shouldn't see any scrapbook pages loaded because we haven't added any yet.

17.3 Databinding events

Earlier in this chapter, you learned how to use Angular's one-way databinding syntax to bind public properties in a component to the UI markup. You also learned that the syntax differed from vanilla NativeScript mustache syntax because it placed square brackets around the UI attributes. For example, to bind the `text` attribute of a `label` element to the `name` field of a component you would use `<Label [text]="name"></Label>`.

Another difference between databinding in vanilla NativeScript and NativeScript-with-Angular apps is the syntax used to bind UI markup events. In vanilla NativeScript, event binding was accomplished

through the same mustache syntax as property binding, but Angular throws you a curve-ball for event binding.

NOTE UI events are bound by placing parenthesis () around the UI event attribute and then placing a function call in the value of the attribute. For example, to bind a button's *tap* event to the *onTap()* function of a component, you use: <Button (tap)="onTap()"></Button>.

Yeah, we're with you. Angular's databinding syntax *does* feel strange. Plus, how are you supposed to keep track of the when to use brackets and when to use parenthesis? Unfortunately, there's no easy way. You'll just have to commit it to memory. But, we have one tip that might help.

TIP Remembering that events need parenthesis in the value and one-way property binding doesn't is frustrating. We found it was easier to remember that parenthesis around the attribute means parenthesis in the value.

17.3.1 Using event binding to add a new scrapbook page

Now that you know the basis of event binding, let's combine it with what you've learned about routing in the previous chapter and add a button to the List component's action bar that navigates to the Detail component. We're missing a few things, like the button, the correct routing configuration, and the details component, so there's a lot to do. Because there are so many steps, we've outlined our plan below:

- Step 1: Add a button to the List component's action bar
- Step 2: Create a tap event handler in the *ListComponent* class
- Step 3: Create the Detail component and register it with the app module
- Step 4: Update the routing configuration to create a Detail component route

Let's go!

STEP 1: ADD A BUTTON TO THE LIST COMPONENT'S ACTION BAR

Get started by adding an action item to the action bar in the list.html file. When you're finished the action bar element should look like listing 17.6.

Listing 17.6 The updated action bar in list.html

```
<ActionBar title="Pet Scrapbook">
  <ActionItem (tap)="onAddTap()" text="Add" //#
    ios.position="right" android.position="actionBar">
  </ActionItem>
</ActionBar>
#A Remember event binding uses parenthesis around the event attribute and in the attribute value
```

You've seen similar UI markup in the original Pet Scrapbook app. The only change is the syntax for binding the *tap* event: *(tap)="onAddTap()"*. By adding parenthesis around the *tap* attribute and placing a function call into the attribute value, Angular expects a public function named *onAddTap()* to exist in the List component.

Yes, it's that quick.

STEP 2: CREATE A TAP EVENT HANDLER IN THE LISTCOMPONENT CLASS

With the event binding configured in the List component's UI mark, let's turn our attention to the `onAddTap()` function in `list.component.ts`. Listing 17.7 shows the updated List component.

Listing 17.7 The `list.component.ts` file with the `onAddTap()` function defined

```
import { Component, OnInit } from "@angular/core";
import { Page } from "../../models/page";
import { RouterExtensions } from "nativescript-angular/router";
import { NavigationOptions } from "nativescript-angular/router/ns-location-strategy";
// #A
import { PageService } from "../../services/page.service";

@Component({
    selector: "list",
    providers: [ PageService ],
    templateUrl: "views/list/list.html",
    styleUrls: ["views/list/list.css"]
})

export class ListComponent implements OnInit {
    public pages: Array<Page>;

    constructor(
        private routerExtensions: RouterExtensions,
        private pageService: PageService) { }

    ngOnInit(): void {
        this.pages = this.pageService.getPages();
    }

    onAddTap(): void {
        let options: NavigationOptions = { // #A
            clearHistory: true // #A
        }; // #A

        this.routerExtensions.navigate(
            ["detail", this.pages.length], // #B
            options);
    }
}

#A NavigationOptions class is needed to clear navigation history
#B To pass data between pages, include it in the routing array as an additional value
```

Adding the `onAddTap()` function is straightforward, but you'll notice something new going on inside of the `navigate()` function: a second value in the routing array. To pass data between components, you can include any number of additional values in the routing array. The first value, `"detail"`, is the route we're navigating to. Any subsequent values (like `this.pages.length`) are appended to the route and passed to the destination component.

Also of note in listing 17.7 is the `NavigationOptions` class.

DEFINITION The *NavigationOptions* class is used to control how navigation between components occurs and settings such as clearing the navigation history, animations, and transitions.

You'll also notice that the *NavigationOptions* class must be imported before we can use it. It's a member of the `@nativescript-angular/router/ns-location-strategy` module and should be added to the import statement as shown in listing 17.7.

NOTE You might be wondering why we're passing in the length of the pages array to the details component. We're using it as a unique identifier for the page we're about to create. You'll recall that each page has a unique identifier, and using the length of the pages array is quick way of getting a unique id for each page. Another method for generating a unique identifier is to read the list of scrapbook pages from the file system, find the highest id, and increment it by one. Putting the methodology of getting a unique identifier aside, what's important is getting one.

STEP 3: CREATE THE DETAIL COMPONENT AND REGISTER IT WITH THE APP MODULE

Now that we've updated the List component, let's turn our attention to the Detail component. We haven't created it yet, so start by creating a folder named `detail` inside the `app` folder, then create three files in the `detail` folder: `detail.html`, `detail.css`, and `detail.component.ts`.

After you've created the component files, add an action bar and action item element to the `detail.html` file, reproducing the same UI from the original Pet Scrapbook (listing 17.8).

NOTE Don't worry if the Detail component looks a bit sparse, we'll come back it later. For now, it will act as a placeholder so we know our routing configuration works.

Listing 17.8 The detail.html file with the onAddTap() function defined

```
<ActionBar [title]="'Page id: ' + page.Id" >
  <ActionItem (tap)="onDoneTap()" text="Done"
    ios.position="right" android.position="actionBar">
  </ActionItem>
</ActionBar>
```

You'll notice in listing 17.8 that the title of the action bar will display the scrapbook page's Id property, which we'll get from the data passed into this component from the routing values. It's not very clear how we'll get this data yet, but we'll cover this shortly, so hang in there.

We also want to point out that the Done button has its tap event bound to the `onDoneTap()` event, like the Add button was bound in the List component.

With the UI markup defined, let's update the `DetailComponent` class. Add the contents of listing 17.9 to the `detail.component.ts` file.

Listing 17.9 The detail.component.ts file loading route value parameters and handling the onDoneTap() event

```
import { Component, OnInit } from "@angular/core";
import { Page } from "../../models/page";
import { RouterExtensions, PageRoute } from "nativescript-angular/router";
```

```

// #A
import { NavigationOptions } from "nativescript-angular/router/ns-location-strategy";
import { PageService } from "../../services/page.service";
import "rxjs/add/operator/switchMap";
// #A

@Component({
    selector: "detail",
    providers: [ PageService ],
    templateUrl: "views/detail/detail.html",
    styleUrls: ["views/detail/detail.css"]
})
export class DetailComponent implements OnInit {
    page: Page; // #B

    constructor(
        private routerExtensions: RouterExtensions,
        private pageService: PageService,
        private pageRoute: PageRoute) { }

    ngOnInit(): void {
        let id:number; // #C
        this.pageRoute.activatedRoute // #C
            .switchMap(activatedRoute => activatedRoute.params) // #C
            .forEach((params) => { // #C
                id = +params["id"]; // #C
            });
        // #C

        this.page = this.pageService.getPage(id); // #D
        if (this.page === null) { // #D
            this.page = <Page>{ Id: id }; // #D
        }
    }

    onDoneTap(): void { // #E
        this.pageService.savePage(this.page);

        var options = <NavigationOptions>{
            clearHistory: true
        };
        this.routerExtensions.navigate(["list"], options);
    }
}

#A PageRoute and the rxjs module are needed to parse the route parameters
#B A scrapbook page is stored in the component for databinding
#C On init, get the id field passed via a route parameter
#D If a page can't be loaded via its id, create a new page with that id
#E When done, save the scrapbook page and navigate back to the List component

```

There's so much going on in the Detail component, it may be a bit confusing. Let's break it down into three areas: getting data passed in via the route parameters, loading a scrapbook page, and saving it when we're done.

GETTING DATA PASSED IN VIA THE ROUTE PARAMETERS

At the top of listing 17.9 is the first evidence of getting data passed in via the route parameters, importing `PageRoute` from `@nativescript-angular/router` and importing `rxjs/add/operator/switchMap`.

DEFINITION The `PageRoute` class is used to get an instance of the `ActivatedRoute` class, an Angular class that holds route parameters.

You don't need to know the intimate details of how and why the `PageRoute` class works. Just remember to import it into components that have data passed into them, then inject an instance of the class via the constructor.

To get the data passed into the `Detail` component, we use the injected page route instance:

```
this.pageRoute.activatedRoute
  .switchMap(activatedRoute => activatedRoute.params)
    .forEach((params) => { id = +params["id"]; });
```

WARNING This code looks and feel overly complicated, and that's because inside the page route instance and its `activatedRoute` property, there's a lot going on with observables. We're not going to cover these details, because it's beyond the scope of this book. But, you may want to learn more about how NativeScript-with-Angular apps navigate between pages. To find out more, check out <https://docs.nativescript.org/core-concepts/angular-navigation#passing-parameter>.

Even though we're not going to dive into the details, it's still important to understand (at a high-level) how to extract parameters from the page route instance. Inside of `PageRoute` is the `activatedRoute` property, which has a `params` object. But, accessing the `params` object isn't easy because of how the `activatedRoute` property works. So, to make it easier to read data from the `activatedRoute` property, the `switchMap()` function is used.

NOTE The `switchMap()` function comes from the imported `rxjs/add/operator/switchMap module`, and in essence creates an array of routing parameter we can read. To learn more about RxJS and the `switchMap()` function, check out <http://reactivex.io/rxjs/class/es6/Observable.js~Observable.html#instance-method-switchMap>.

Once we have an array of parameters from the `switchMap()` function, we iterate through them and grab the `id` value.

TIP You may have noticed the strange `+` sign in listing 17.9: `id = +params["id"];`. It's called a *unary operator*, and performs a datatype conversion from a string held in `params["id"]` to a number. We wanted to point this out because you may have never run across it before, and spent hours trying to Google "plus sign TypeScript."

LOADING A SCRAPBOOK PAGE

After getting the scrapbook page `id`, the page service is used to load the scrapbook page (stored in the `this.page` property of the `Detail` component):

```
this.page = this.pageService.getPage(id);
if (this.page === null) { this.page = <Page>{ Id: id }; }
```

If the page service returns null, that means there is no existing scrapbook page with the referenced *id*, so a new page is created and stored in the *this.page* property.

SAVING THE SCRAPBOOK PAGE

The final section of code we want to review in listing 17.9 is the *onDoneTap()* function, which is bound to the Done action item button. When we're finished entering data, *this.pageService.sagePage(this.page)* is called to save the scrapbook page. After the page has been saved, we navigate back to the List component.

STEP 4: UPDATE THE ROUTING CONFIGURATION TO CREATE A DETAIL COMPONENT ROUTE

After updating the Detail component file, the last item that needs done is to configure a route for the Detail component in the *app.routing.ts* file. Listing 17.10 outlines the changes made.

Listing 17.10 The app.routing.ts file updated to include a route for the Detail component

```
import { HomeComponent } from "./views/home/home.component";
import {ListComponent} from "./views/list/list.component";
import {DetailComponent} from "./views/detail/detail.component"; // #A

export const routes: any = [
  { path: "", component: HomeComponent },
  { path: "home", component: HomeComponent },
  { path: "list", component: ListComponent },
  { path: "detail/:id", component: DetailComponent } // #A
];

export const navigatableComponents: any = [
  HomeComponent,
  ListComponent,
  DetailComponent // #A
];
#A To add the Detail component route, import it, add it to the routes and component arrays
```

Adding the detail route is just like adding the route for the list and Home components, with one exception: the detail route also needs to include a placeholder for the *id* parameter we'll be passing in:

```
{ path: "detail/:id", component: DetailComponent }
```

NOTE Route parameters are declared using the `/:{parameter-name}` syntax following the route name. For example, the route path of "*detail/:id*" describes a route named *detail* with a single parameter named *id*. Even though this example only shows one route parameter, additional parameters can be added.

Route parameters are a standard Angular functionality, so we're not going to dive any deeper on the topic. If you're still interested in learning more, check out official NativeScript-with-Angular documentation at <https://docs.nativescript.org/core-concepts/angular-navigation#navigation>.

With these changes to the Pet Scrapbook Angular app, you should be able to create new scrapbook pages and navigate back to the List component. If you're following along, your version of the app should look like figure 17.4.

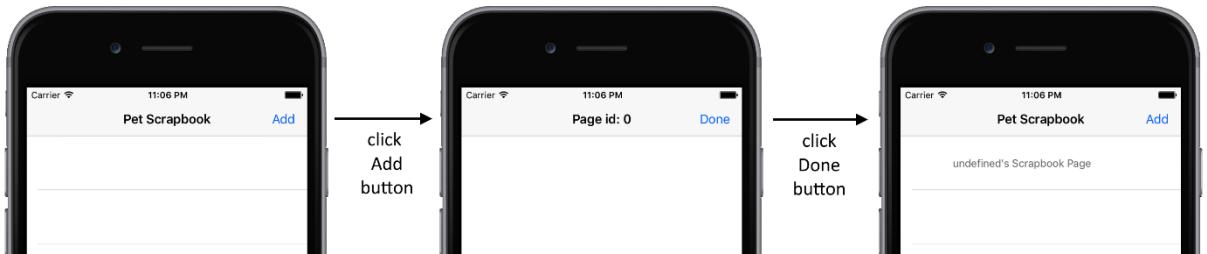


Figure 17.4 After saving a scrapbook page, it is displayed on the List component.

It may not look pretty, but it's functional. Tapping the Add button navigates to the Detail component. When finished, tapping the Done button saves the scrapbook page and navigates back to the List component, showing the saved page.

But, what happens when you tap the scrapbook page in the list? Nothing right now. Let's continue to explore event data binding by adding a tap handler for an item in the list.

17.3.2 Passing data with event databinding

In the last section, you learned how to use Angular's databinding syntax to bind UI element events to component functions. The examples from the Pet Scrapbook Angular app introduced simple event binding.

DEFINITION Simple event binding is where an event handler doesn't need to know any information (or context) about the data bound UI element to perform its intended purpose. For example, tapping the Continue button on the Home component and the Add button on the List component are examples of simple event binding: to navigate to another component, it doesn't need to know anything about its current state.

Not all UI markup events are simple events, and they often need to know about their surroundings. Take the list of scrapbook pages on the List component as an example. In the original Pet Scrapbook app, when you tapped on an item in the list, you navigated to a detail page for that item. This is an example of complex event binding: the handler for the list item tap event needed information (like the scrapbook page id of the tapped list item) to navigate to the detail page and display the correct scrapbook page.

It turns out that passing data (or context) to an event handler is easy in NativeScript-with-Angular apps. Let's investigate how it's done with Angular's databinding syntax by updating the list view in the *list.html* file to add the *itemTap* event:

```
<ListView class="list-group" [items]="pages" (itemTap)="onItemTap($event)">
```

As you can see, Angular does indeed make it easy. Binding to an event is the same syntax, and to pass along the event data (or context) to the code behind, we added the *\$event* keyword. The *\$event* keyword passes event data to the tap event handler automatically.

HANDLING EVENTS WITH EVENT DATA

Handling events bound with the `$event` keyword is just like handling simple data-bound events. Let's start by adding the event handler code from listing 17.11 to the `ListComponent` class.

Listing 17.11 The list.component.ts file updated to include the itemTap event handler

```
import { ItemEventData } from "ui/list-view";

onItemTap(args: ItemEventData): void { // #A
    let id = args.index;

    this.routerExtensions.navigate(["detail", id]);
}

#A the $event keyword injects a parameter into the event handler
```

When the `$event` keyword is used in databinding an event, it injects event data into the event handler as a parameter. The datatype of the injected parameter varies depending on the event type. There's not enough room to outline every UI event here, and you don't need to memorize them because it's easy to look up online. Check out the official NativeScript API at <https://docs.nativescript.org/api-reference/globals.html> for more information.

And that's it! We said it was easy. If you've been following along, you can now tap a scrapbook page to navigate to the detail page. When the Detail component loads, it will parse the page's id, load the page from the file system, then display the data-bound UI. Figure 17.5 shows the Pet Scrapbook Angular app after making this change.

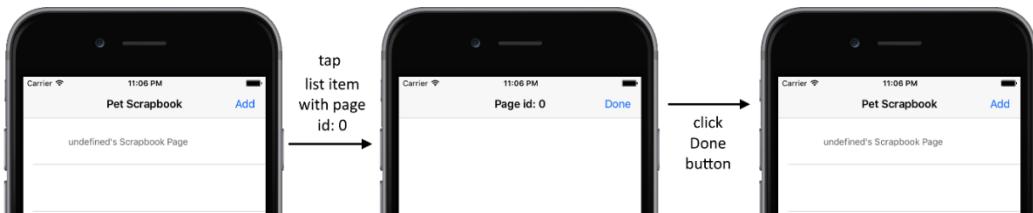


Figure 17.5 Tapping a list item navigates to the details component while passing along the item's page id.

17.4 Advanced databinding

Now that you've learned the basics of databinding by using the one-way and event databinding syntaxes, it's time to step up our game and learn various advanced databinding techniques that can be used in NativeScript-with-Angular apps.

Before we get ahead of ourselves, let's start by updating the Detail component's `detail.html` file with the code in listing 17.12. For now, we'll be adding in one-way data-binding for the UI elements, then we'll come back and make a few minor changes to use two-way databinding.

Listing 17.12 The detail.html file updated for one-way databinding using Angular

```
<ActionBar [title]="page.Title === null || page.Title === undefined || // #A
page.Title === '' ? 'New Page' : page.Title + '\s Page'"> // #A
```

```

<ActionBar text="Done" (tap)="onDoneTap()"
  ios.position="right" android.position="actionBar">
</ActionBar>
</StackLayout>
<StackLayout class="form">
  <StackLayout class="input-field">
    <Label class="label" text="Name:" >
    </Label>
    <TextField class="input" [text]="page.Title" hint="Enter a name..."> // #A
    </TextField>
  </StackLayout>
  <StackLayout class="input-field">
    <Label class="label" [text]="'Birth date: ' +
      (page.BirthDate === null ? '' : '(' + page.Age + ' years old)')"> // #B
    </Label>
    <TextField class="input" editable="false"
      [text]="page.BirthDate"
      (tap)="onBirthDateTap()"
      hint="Enter a birth date..."> // #B
    </TextField>
  </StackLayout>
  <StackLayout class="input-field">
    <Label class="label" text="Gender:" >
    </Label>
    <TextField class="input" editable="false"
      (tap)="onGenderTap()"
      [text]="page.Gender" hint="Select a gender..."> // #C
    </TextField>
  </StackLayout>
  <StackLayout class="input-field">
    <Label class="label" text="Image:" >
    </Label>
    <Image [src]="page.Image" stretch="None"> // #D
    </Image>
    <Label class="footnote" [text]="(page.Lat === undefined ||
      page.Long === undefined) ? '' : 'Picture taken at ' +
      page.Lat + ', ' + page.Long"> // #E
    </Label>
  </StackLayout>
</StackLayout>
<Button text="Add Image" (tap)="onAddImageTap()"
  class="btn btn-primary btn-rounded-sm btn-active">
</Button>
</StackLayout>
#A Binds the title
#B Binds the birthdate and age
#C Binds the gender
#D Binds the image
#E Binds the lat and long

```

We're not going to review listing 17.12 in detail because it's the same UI elements we used in the original Pet Scrapbook.

NOTE You'll notice that we haven't defined the event handlers for the birth date, gender, and image buttons. Don't worry, we didn't forget. We'll look at those later in the chapter.

Listing 17.12 is a great starting place, because it shows one-way databinding for the details component, but as you learned in previous chapters, one-way isn't the only way (especially if we expect users to enter data). It's time to check out two-way databinding.

17.4.1 Two-way databinding

Two-way databinding in Angular apps uses something called *ngModel*.

DEFINITION *ngModel* is a special Angular directive that allows data to flow in two directions: from a property to the UI markup, then from the UI markup back to the property.

Before you jump to any conclusions, using *ngModel* is easy, and there's a minor syntax change you need to learn to use it.

Unfortunately, support for *ngModel* isn't available unless you import the *NativeScriptFormsModule*, but that's easily remedied. Let's get started using *ngModel* by updating the app module as shown in listing 17.13.

Listing 17.13 The app.module.ts file updated to use the NativeScript forms module

```
import { NgModule, NO_ERRORS_SCHEMA } from "@angular/core";
import { NativeScriptModule } from "nativescript-angular/nativescript.module";
import { NativeScriptFormsModule } from "nativescript-angular/forms"; // #A
import { NativeScriptRouterModule } from "nativescript-angular/router";
import { routes, navigatableComponents } from "./app.routing";

import { AppComponent } from "./app.component";
import { HomeComponent } from "./views/home/home.component";
import {ListComponent} from "./views/list/list.component";
import { DetailComponent } from "./views/detail/detail.component";

@NgModule({
  declarations: [
    AppComponent,
    ...navigatableComponents
  ],
  bootstrap: [
    AppComponent
  ],
  imports: [
    NativeScriptModule,
    NativeScriptFormsModule, // #A
    NativeScriptRouterModule,
    NativeScriptRouterModule.forRoot(routes)
  ],
  schemas: [NO_ERRORS_SCHEMA],
})

export class AppModule { }

#A Import the forms module to enable two-way databinding
```

Two minor changes were made to import the *NativeScriptFormsModule* into the app module.

NOTE If you're familiar with Angular, you'll be familiar with the Angular module named *FormsModule*, which is the default Angular module used to perform two-way databinding with *ngModel*. In NativeScript-with-Angular apps, we use a NativeScript-specific version of the forms module, *NativeScriptFormsModule*, which integrates the Angular *FormsModule* with NativeScript.

After importing this module, implementing two-way databinding is a simple change to the UI code; in fact, it is just another syntax difference. For example, in one-way databinding, to bind the text of a label to the name property of a component, we would use `[text]="name"`. When using *ngModel*, `[text]` gets replaced with `[(ngModel)]`, creating `[(ngModel)]="name"`.

Before we go any further, let's update the `detail.html` file to use *ngModel* for the text field bound to the title property. Change the text field to look like this:

```
<TextField class="input" [(ngModel)]="page.Title" hint="Enter a name...">
</TextField>
```

Reflecting back, the text field bound the text attribute `[text]="page.Title"`, which was one-way databinding. With two-way databinding, `[(ngModel)]` is used instead of `[text]`.

NOTE The `[(ngModel)]` syntax may feel strange, but there is a method to the madness. The syntax is really a shorthand for the following: `[text]="$title" (ngModelChange)="title=$event"`. Let that settle in for a minute. `[text]="$title"` does one-way databinding from the component property to the UI layer. Although we haven't discussed *(ngModelChange)* specifically, it uses the parenthesis format, which is an event. That means that there's an event called *ngModelChange*, and when it is raised, we should call the code it specifies (`title=$event`). *ngModelChange* is raised when the UI element updates, so it acts like another one-way databinding, except it's from the UI to code. And guess what? When you have two one-way databinding processes working in concert, it's considered two-way.

Let's check out the two-way databinding in action by running the Pet Scrapbook Angular app. You'll notice as you update the name text field, action bar text also updates (figure 17.6).

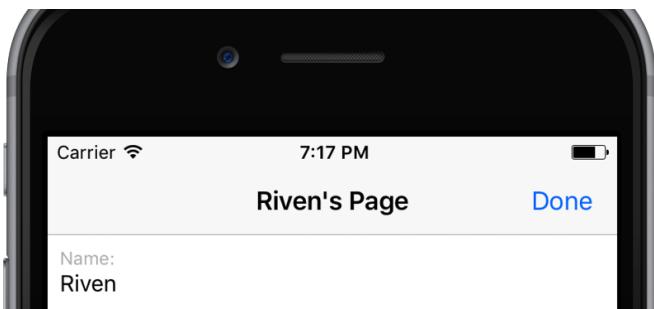


Figure 17.6 When text is entered into the name text field, the action bar text also updates.

Using `ngModel` is straightforward, but the first time we used `ngModel`, we had a lingering thought, "How does it know which property to bind to?" It's not magic, but instead a collection of well-known defaults for UI elements that need to participate in two-way binding. For example, when `ngModel` is used in text fields, it assumes you want to bind it to the `text` attribute. Why? Because that's the only attribute where data is entered in, and you don't need two-way databinding on fields that don't support data entry.

If you're still in doubt as to which fields `ngModel` applies to on each element, you can use table 17.1 as a reference.

Table 16.1 The NativeScript UI elements that support the `ngModel` syntax and the properties that Angular binds to

UI Element	Property
DatePicker	date
ListPicker	selectedIndex
SearchBar	text
SegmentedBar	selectedIndex
Slider	value
Switch	checked
TextField	text
TextView	text
TimePicker	time

17.4.2 Formatting data-bound properties

The final advanced databinding technique we're going to cover for NativeScript-with-Angular apps is formatting data-bound properties. If you can recall, the original Pet Scrapbook used a databinding expression to format the birth date: `text="{{ birthDate, birthDate | dateConverter(dateFormat) }}"`. This expression calls a globally-registered function, `dateConverter()`, to provide the formatting.

In NativeScript-with-Angular apps, there's a similar functionality, but it's much easier. Angular has a set of formatting functions for dates. Let's use one by updating the birth date text field's data-bound property. Use the pipe (`|`) operator and a date formatting function:

```
[text]="page.BirthDate | date:'shortDate'"
```

This expression binds the `text` attribute to the `birth date` property of the `page` object, and applies a short date formatting function to it. When displayed, it will show `mm/dd/yyyy`. The short date function

isn't the only one available. To learn more about the formatting options that are available see <https://angular.io/docs/ts/latest/api/common/index/DatePipe-pipe.html>.

That's all we're going to cover about databinding in a NativeScript-with-Angular apps, because that's everything you'll need to know to start building your first app. If you're still looking for more, we recommend checking out the official NativeScript documentation at <http://docs.nativescript.org/angular/core-concepts/angular-data-binding.html>.

17.5 Loading components as modal dialogs

In chapter 11, you learned how *modal dialogs* can be used to improve the user experience of the Pet Scrapbook Angular app. In case you forgot, modal dialogs (or modals) are a UI design concept where a user's interaction on a page prompts the UI to temporarily display a second page on top of the first page. When the user completes their interaction with the second page, it disappears and the UI is redirected to the first page.

We set out to add modals to the Pet Scrapbook initially because we wanted to hide the complexity of the birth date and gender selection UIs from the detail page. Figure 17.7 is a reminder of the workflow we introduced.

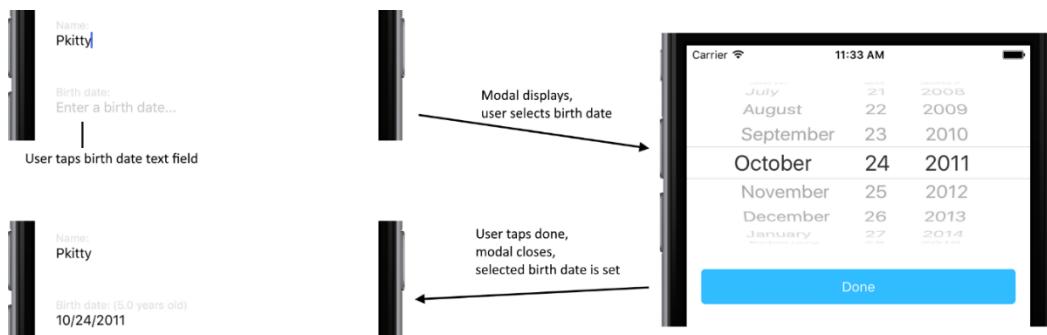


Figure 17.7 A user's interaction with the update page with modal dialogs used to select the birth date.

We'll be using the same workflow in the NativeScript-with-Angular version of our app, so keep it in mind as you learn to use modals in this section.

17.5.1 Adding a date selection modal

If you recall from chapter 11, modals are simply pages opened in a different way. So, as we explore modals in a NativeScript-with-Angular app, we'll be using components.

NOTE Remember, NativeScript-with-Angular app components are equivalent to vanilla NativeScript pages. And if a modal is a page in vanilla NativeScript apps, a modal is a component in NativeScript-with-Angular apps.

Putting aside the difference that modals are components, there's a different method used to create and display a modal. Let's see it in action by adding a modal to select the birth date for a scrapbook page. The modal is really a component and will be called `select-date` (yeah, real creative name there...).

Start by creating a `modals` folder in the `views` folder, then add a folder named `select-date`. Create the three `select-date` component folders in this folder: `select-date.html`, `select-date.css`, and `select-date.component.ts`.

NOTE Notice the naming convention of the `select-date` component. Components (and various other entities like services) use a dash in the file name when the name is compound (like `select-date`). This is an Angular-specific style guide rule outlined online at <https://angular.io/styleguide>.

Add the contents of listing 17.14 to the `select-date.html` file.

Listing 17.14 The `select-date.html` file that defines the UI of the `select-date` component (and modal)

```
<StackLayout>
    <DatePicker [(ngModel)]="date"></DatePicker> // #A
    <Button class="btn btn-primary btn-rounded-sm btn-active"
        text="Done" (tap)="onDoneTap()" >
    </Button>
</StackLayout>
#A DatePicker elements can be bound with ngModel
```

The UI markup for the `select-date` component is quite simple. One item to note that you'll recognize from earlier in this chapter is the date picker element is bound using `ngModel` to enable two-way databinding.

Next, update the `select-date.component.ts` file, adding the content of listing 17.15.

Listing 17.15 The `select-date.component.ts` file that defines the logic and functionality

```
import { Component } from "@angular/core";
import { ModalDialogParams } from "nativescript-angular/modal-dialog"; // #A

@Component({
    selector: "select-date",
    templateUrl: "views/modals/select-date/select-date.html",
    styleUrls: ["views/modals/select-date/select-date.css"]
})
export class SelectDateComponent {
    date: any;

    constructor(private params: ModalDialogParams) { // #B
        this.date = params.context; // #B
    } // #B

    onDoneTap(): any {
        this.params.closeCallback(this.date); // #C
    }
} // #A To read parameters passed into modals, you need ModalDialogParams
  // #B The data-bound date property is set to an initial value, passed in via a parameter
```

#C The modal returns the selected date when it closes

Let's dissect what's going on in listing 17.15. First, we've imported the *ModalDialogParams* class from the *nativescript-angular/modal-dialog* module.

DEFINITION The *ModalDialogParams* class is a special class responsible for tracking any data (or parameters) passed into the modal when it's opened.

Although we haven't seen the code that opens the modal dialog yet, we can surmise that the current value for the birth date is passed in as a parameter. To read that value, the *context* property of the *ModalDialogParams* class instance is used. When the passed in parameter value is read, it gets assigned to the *date* property. But you'll notice the *date* property is declared as a datatype of *any*, which seems a bit strange.

TIP The date picker UI element can't be initialized automatically if the datatype of the data-bound property is a *Date*. As strange as this seems, you can trick the UI element to bind by declaring the property as a type of *any*. Unfortunately, using the *any* datatype removes the type-safe advantage of TypeScript, but we're willing to accept that in this case. To see an alternate way of databinding with the *Date* datatype, check out the official NativeScript documentation at <https://docs.nativescript.org/angular/code-samples/modal-page>.

The last point of interest is the Done button's tap event handler, *onDoneTap()*. If you recall, vanilla NativeScript modals are closed by calling a close callback function. NativeScript-with-Angular apps work the same way: the *ModalDialogParams* class has a property called *closeCallback*, which is a function that gets called to pass data back to the calling component and close the dialog: `this.params.closeCallback(this.date);`.

REGISTERING THE SELECT-DATE COMPONENT AS AN ENTRY COMPONENT
Before we can use the select-date component as a modal, we need to update the app module by registering the components as *entry components*.

DEFINITION In the simplest of terms, entry components are a collection of components that are dynamically loaded through code, and not routed to via navigation. Modal components fall into this category, so you need to register them in a special way. We're not going to dive deeper on entry components, because they're a much more complex topic. Just remember to register any modals in the app module. If you're interested in learning more about entry components, check out <https://angular.io/docs/ts/latest/cookbook/ngmodule-faq.html#!#q-entry-component-defined>.

To register the modals, import them into the app module and add an *entryComponents* property to the *@NgModule* declaration. Listing 17.16 shows these updates to the *app.module.ts* file.

Listing 17.16 The *app.module.ts* file updated to include the select-date modal

```
import { NgModule, NO_ERRORS_SCHEMA } from "@angular/core";
```

```

import { NativeScriptModule } from "nativescript-angular/nativescript.module";
import { NativeScriptFormsModule } from "nativescript-angular/forms"
import { NativeScriptRouterModule } from "nativescript-angular/router";
import { routes, navigatableComponents } from "./app.routing";

import { AppComponent } from "./app.component";
import { HomeComponent } from "./views/home/home.component";
import { DetailComponent } from "./views/detail/detail.component";
import { SelectDateComponent } from "./views/modals/select-date/select-date.component"; // #A
import { ListComponent } from "./views/list/list.component"; // #A

@NgModule({
  declarations: [
    AppComponent,
    SelectDateComponent, // #B
    ...navigatableComponents
  ],
  entryComponents: [ // #B
    SelectDateComponent // #B
  ], // #B
  bootstrap: [
    AppComponent
  ],
  imports: [
    NativeScriptModule,
    NativeScriptFormsModule,
    NativeScriptRouterModule,
    NativeScriptRouterModule.forRoot(routes),
  ],
  schemas: [NO_ERRORS_SCHEMA],
})
export class AppModule {
}

#A Import the select-date modal
#B Declare and register it in the entryComponents property

```

UPDATING THE DETAIL COMPONENT TO OPEN THE MODAL

The final step to using the select-date modal is to update the Detail component's code file to handle the tap event of the birth date text box UI element. You'll recall we already added the databinding express to the text box:

```

<TextField [text]="page.BirthDate | date:'shortDate'"
  class="input" editable="false"
  (tap)="onBirthDateTap()" hint="Enter a birth date..." ></TextField>

```

To open the select-date modal when the text box is tapped, update the detail.component.ts file as shown in listing 17.17.

Listing 17.17 The detail.component.ts file updated to include code to open the select-date modal

```

import { Component, OnInit, ViewContainerRef } from "@angular/core"; // #A
import { Page } from "../../models/page";
import { RouterExtensions, PageRoute } from "nativescript-angular/router";

```

```
import { NavigationOptions } from "nativescript-angular/router/ns-location-strategy";
import { PageService } from "../../services/page.service";
import "rxjs/add/operator/switchMap";
import { ModalDialogService, ModalDialogOptions } // #A
    from "nativescript-angular/modal-dialog";           // #A
import { SelectDateComponent } from "../modals/select-date/select-date.component"; // #A

@Component({
    selector: "detail",
    providers: [ PageService ],
    templateUrl: "views/detail/detail.html",
    styleUrls: ["views/detail/detail.css"]
})
export class DetailComponent implements OnInit {
    page: Page;

    constructor(
        private routerExtensions: RouterExtensions,
        private pageService: PageService,
        private pageRoute: PageRoute,
        private modalService: ModalDialogService,           // #B
        private viewContainerRef: ViewContainerRef) { } // #B

    ngOnInit(): void {
        let id:number;
        this.pageRoute.activatedRoute
            .switchMap(activatedRoute => activatedRoute.params)
            .forEach((params) => {
                id = +params["id"];
            });

        this.page = this.pageService.getPage(id);
        if (this.page === null) {
            this.page = <Page>{ Id: id };
        }
    }

    onDoneTap(): void {
        this.pageService.savePage(this.page);

        var options = <NavigationOptions>{
            clearHistory: true
        };
        this.routerExtensions.navigate(["list"], options);
    }

    onBirthDateTap(): void {
        let options: ModalDialogOptions = {                  // #C
            context: this.page.BirthDate,                  // #C
            fullscreen: true,                            // #C
            viewContainerRef: this.viewContainerRef // #C
        };                                                 // #C

        this.modalService.showModal(SelectDateComponent, options) // #D
            .then((dialogResult: any) => {               // #E
                this.page.BirthDate = dialogResult;       // #E
            })
    }
}
```

```

        let now = Date.now(); // #E
        let diff = Math.abs(now - this.page.BirthDate) / 1000 / 31536000; // #E

        this.page.Age = diff.toFixed(1); // #E
        // #E
    });
}

#A To open modals, we need to import several modules
#B The ModalService and ViewContainerRef modules need injected
#C The ModalDialogOptions has various properties used to configure how a modal is opened and the data
passed in
#D Modals are opened by passing the modal component class into showModal()
#E When the close callback is called, the code in then() is executed

```

A lot was added to the Detail component, so let's start at the top with some new imports: *ModalService*, and *ModalDialogOptions*, and *ViewContainerRef*. *ModalService* is used to open the modal dialog, and *ModalDialogOptions* contains various properties that configure how the dialog is opened and what data is passed to the dialog. *ViewContainerRef* is a little more complicated, but essential.

DEFINITION *ViewContainerRef* is an Angular class that can track an Angular view. The exact details of *ViewContainerRef* aren't necessary for you to understand. We like to think of it as the reference to the current component, and it's used by the opened modal to know where to go back to when the modal is closed. To learn more about this class, check out the Angular documentation at <https://angular.io/docs/ts/latest/api/core/index/ViewContainerRef-class.html>.

In the constructor of the Detail component, you'll notice that *ModalService* and *ViewContainerRef* instances are being injected.

Next, the *onBirthDateTap()* function is added to handle the tap event for the birth date text box. A set of modal dialog options is created using the *ModalDialogOptions* class:

```

let options: ModalDialogOptions = {
    context: this.page.BirthDate,
    fullscreen: true,
    viewContainerRef: this.viewContainerRef
};

```

The context property is used to pass data into the modal dialog, and you can see that we pass in the pet's currently selected birth date. You'll also see where the *ViewContainerRef* class is passed to the modal dialog, like we explained earlier.

To open the select-date modal, we use the *ModalService* class instance injected into the constructor and call the *showModal()* function: `this.modalService.showModal(SelectDateComponent, options)`.

NOTE The *showModal()* function is interesting because the first parameter isn't an instance of a class, but a reference to a class. It uses the reference to dynamically instantiate the *SelectDateComponent* when called.

Attached to the `showModal()` function is another code block that runs when the select-date's close callback function is called: `.then((dialogResult: any) => { ... })`. The `dialogResult` variable contains the result passed back through the close callback function.

NOTE You may not recognize the `.then(() => {})` syntax attached to the `showModal()` function, which is a way to asynchronously run code when the `showModal()` function finishes running and the dialog is closed. We can do this because `showDialog()` returns a promise. We've talked about promises before, so we won't go into detail here. If you need a refresher, check out Mozilla's documentation on promises at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.

Now that we've finished updating the app to use the select-date dialog, let's run the Pet Scrapbook Angular app. Figure 17.8 shows how the app navigates from the Detail component to the modal, then back.

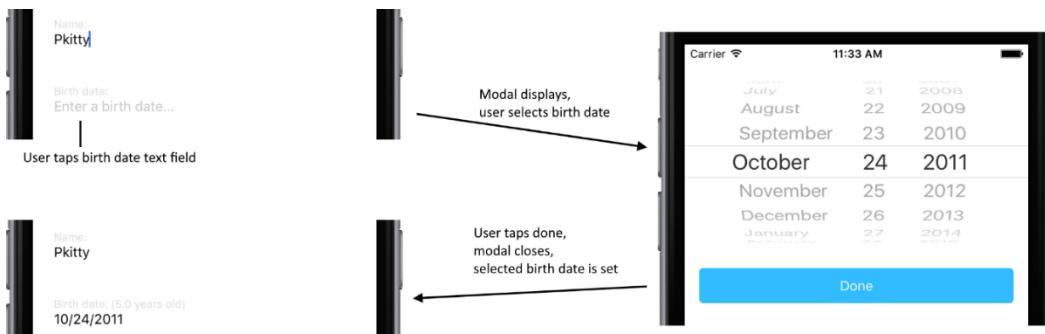


Figure 17.8 The Pet Scrapbook Angular app after adding the select-date modal to select a birth date.

17.5.2 Adding a gender selection modal

Now that you've learned how to add a modal to the Pet Scrapbook Angular app, let's do the same for the gender text box. Start by creating a `select-gender` component in the `modals` folder. When finished, you should have a `select-gender` folder with three files in it: `select-gender.html`, `select-gender.css`, and `select-gender.component.ts`.

Add the code from listing 17.18 to the `select-gender.html` file.

Listing 17.18 The `select-gender.html` file defines the UI markup for the `select-gender` component

```
<StackLayout>
  <ListPicker [items]="genders" [(ngModel)]="gender"></ListPicker> // #A
  <Button class="btn btn-primary btn-rounded-sm btn-active"
    text="Done" (tap)="onDoneTap()">
  </Button>
</StackLayout>
#A List picker is bound to the selected gender property and to the collection of genders
```

The UI markup for the select-gender component is like the select-date component, except we've substituted a list picker so we can select from the list of genders. You'll notice that we're using both one-way and two-way databinding on the element. The *items* attribute is data-bound to the *genders* property (*[items] = "genders"*) using one-way databinding, and the selected item is data bound to the *gender* property (*[(ngModel)] = "gender"*) using *ngModel* and two-way databinding.

Next, define the *SelectGenderComponent* class by adding the code from listing 17.19 to the *select-gender.component.ts* file.

Listing 17.19 The select-gender.component.ts file

```
import { Component } from "@angular/core";
import { ModalDialogParams } from "nativescript-angular/modal-dialog";

@Component({
    selector: "select-gender",
    templateUrl: "views/modals/select-gender/select-gender.html",
    styleUrls: ["views/modals/select-gender/select-gender.css"]
})

export class SelectGenderComponent {
    gender: number;
    genders: Array<string> = ["Female", "Male", "Other"]; // #A

    constructor(private params: ModalDialogParams) {
        this.gender = params.context;
    }

    onDoneTap(): any {
        this.params.closeCallback(this.genders[this.gender]); // #B
    }
}

#A The genders is a static array of genders, data-bound to the items attribute of the list picker
#B When the dialog closes, the selected gender is passed back to the Detail component
```

Just like the select-date component, the select-gender component injects the *ModalDialogParams* class and retrieves the gender passed into the modal when it opens. The only change to point out is how the *genders* property is created: it's an array of strings. This is then bound to the *items* attribute of the list picker, rendering three options to choose from: Female, Male, and Other. After the user picks a gender, the close callback is called, passing back the selected value from the array.

The final two steps are to register the select-gender component as an entry component and open the modal from the details component. Listings 17.20 and 17.21 show how the *app.module.ts* and *detail.component.ts* files are updated.

Listing 17.20 The app.module.ts updated to register the select-gender modal as an entry component

```
import { NgModule, NO_ERRORS_SCHEMA } from "@angular/core";
import { NativeScriptModule } from "nativescript-angular/nativescript.module";
import { NativeScriptFormsModule } from "nativescript-angular/forms"
import { NativeScriptRouterModule } from "nativescript-angular/router";
import { routes, navigatableComponents } from "./app.routing";
```

```

import { AppComponent } from "./app.component";
import { HomeComponent } from "./views/home/home.component";
import { DetailComponent } from "./views/detail/detail.component";
import { SelectDateComponent }
  from "./views//modals/select-date/select-date.component";
import { SelectGenderComponent }
  from "./views//modals/select-gender/select-gender.component"; // #A
import {ListComponent} from "./views/list/list.component"; // #A

@NgModule({
  declarations: [
    AppComponent,
    SelectDateComponent,
    SelectGenderComponent, // #B
    ...navigatableComponents
  ],
  entryComponents: [
    SelectDateComponent,
    SelectGenderComponent // #B
  ],
  bootstrap: [
    AppComponent
  ],
  imports: [
    NativeScriptModule,
    NativeScriptFormsModule,
    NativeScriptRouterModule,
    NativeScriptRouterModule.forRoot(routes),
  ],
  schemas: [NO_ERRORS_SCHEMA],
})
export class AppModule {
}

#A Import the modal
#B Declare and register the modal as an entry component

```

Listing 17.21 The detail.component.ts file updated to include code to open the select-gender modal

```

import { Component, OnInit, ViewContainerRef } from "@angular/core";
import { Page } from "../../models/page";
import { RouterExtensions, PageRoute } from "nativescript-angular/router";
import { NavigationOptions } from "nativescript-angular/router/ns-location-strategy";
import { PageService } from "../../services/page.service";
import "rxjs/add/operator/switchMap";
import { ModalDialogService, ModalDialogOptions }
  from "nativescript-angular/modal-dialog";
import { SelectDateComponent } from "../modals/select-date/select-date.component";
import { SelectGenderComponent }
  from "../modals/select-gender/select-gender.component"; // #A

@Component({
  selector: "detail",
  providers: [ PageService ],
  templateUrl: "views/detail/detail.html",
})

```

```
    styleUrls: ["views/detail/detail.css"]
})
export class DetailComponent implements OnInit {
  page: Page;

  constructor(
    private routerExtensions: RouterExtensions,
    private pageService: PageService,
    private pageRoute: PageRoute,
    private modalService: ModalDialogService,           // #B
    private viewContainerRef: ViewContainerRef) { } // #B

  ngOnInit(): void {
    let id:number;
    this.pageRoute.activatedRoute
      .switchMap(activatedRoute => activatedRoute.params)
      .forEach((params) => {
        id = +params["id"];
      });

    this.page = this.pageService.getPage(id);
    if (this.page === null) {
      this.page = <Page>{ Id: id };
    }
  }

  onDoneTap(): void {
    this.pageService.savePage(this.page);

    var options = <NavigationOptions>{
      clearHistory: true
    };
    this.routerExtensions.navigate(["list"], options);
  }

  onBirthDateTap(): void {
    let options: ModalDialogOptions = {                // #C
      context: this.page.BirthDate,                  // #C
      fullscreen: true,                            // #C
      viewContainerRef: this.viewContainerRef       // #C
    };

    this.modalService.showModal(SelectDateComponent, options)
      .then((dialogResult: any) => {
        this.page.BirthDate = dialogResult;

        let now = Date.now();
        let diff = Math.abs(now - this.page.BirthDate) / 1000 / 31536000;

        this.page.Age = diff.toFixed(1);
      });
  }

  onGenderTap(): void {
    let options: ModalDialogOptions = {
      context: this.page.Gender,
      fullscreen: true,
```

```
    viewContainerRef: this.viewContainerRef
};

this.modalService.showModal(SelectGenderComponent, options) // #D
  .then((dialogResult: string) => {           // #E
    this.page.Gender = dialogResult;          // #E
  });
}

#A To open modals, we need to import several modules
#B The ModalService and ViewContainerRef modules need injected
#C The ModalDialogOptions has various properties used to configure how a modal is opened and the data
passed in
#D Modals are opened by passing the modal component class into showModal()
#E When the close callback is called, the code in then() is executed
```

With the code changes made to the select-gender component, Detail component, and app module, it's time to run the Pet Scrapbook Angular app again. When you tap the gender text box, the select-gender modal component will show, allowing you to select a gender (figure 17.9).

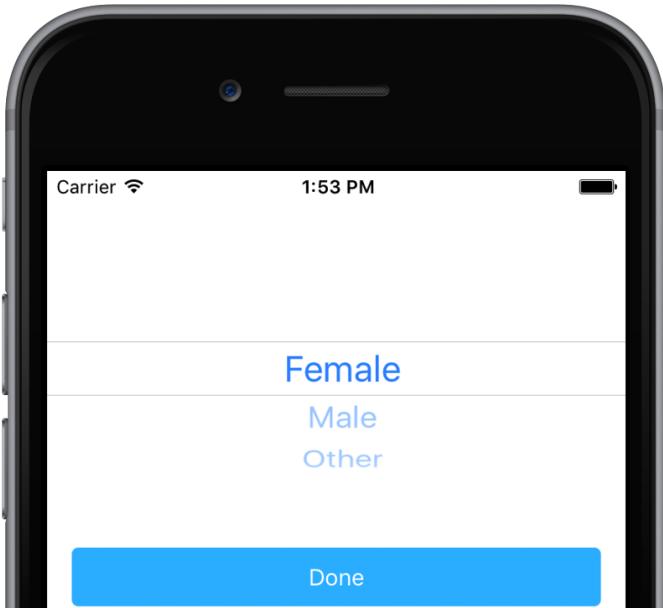


Figure 17.9 Tapping the gender text box on the details component now opens the select-gender component.

And there you have it: we've re-created almost every aspect of the Pet Scrapbook using Angular. You'll notice we left out taking a picture using the camera, but that's on purpose. You've learned everything you need to add that feature into the NativeScript-with-Angular version, so our challenge to you is to add that

into the app. If you get stuck, feel free to check out our GitHub repo at <https://github.com/mikebranstein/TheNativeScriptBook/tree/master/Chapter17/PetScrapbookAngular>.

17.6 Summary

In this chapter, you learned the following:

- How to bind properties and events with one-way databinding, and how two-way (`ngModel`) databinding differs in syntax and functionality
- How service classes (and other Angular classes) are injected into constructors using dependency injection
- How to pass data between components when navigating by using the `RoutingExtensions` class
- How the `PageRoute` class and RxJS module can be used to read routing parameters when a component is initialized
- That modal dialog components must be registered as entry components in the app module

17.7 Exercise

Using what you've learned about databinding, implement the Add Image button tap event handler, which uses the camera module to take a picture using native hardware.

17.8 Solutions

To add in the camera functionality, import the `camera` and `nativescript-geolocation` modules into the `detail.component.ts` file:

```
import * as camera from "nativescript-camera";
import * as geolocation from "nativescript-geolocation";
```

NOTE Remember that the geolocation NPM module is a NativeScript plugin. To install the plugin, you will need to run `tns plugin add nativescript-geolocation` and `tns plugin add nativescript-camera` in the CLI.

Now that we have imported the necessary modules, add the Add Image button tap event handler as shown in listing 17.22.

Listing 17.22 The Add Image button tap event handler to be added to the detail.component.ts file

```
import { ImageSource } from "image-source";

onAddImageTap(): void { // A
    if (!geolocation.isEnabled()) {
        geolocation.enableLocationRequest();
    }

    camera.takePicture({ width: 100, height: 100, keepAspectRatio: true })
        .then((picture) => {
            let image = new ImageSource();
```

```
image.fromAsset(picture).then((imageSource) => {
    this.page.Image = imageSource;
    this.page.ImageBase64 = this.page.Image.toBase64String("png");
});

geolocation.getCurrentLocation(null)
    .then((location) => {
        this.page.lat = location.latitude;
        this.page.long = location.longitude;
    });
});
```

Finally, if you're working on iOS, you'll need to allow camera permissions by updating the app/app_Resources/info.plist file and adding the following keys:

```
<key>NSCameraUsageDescription</key>
<string>This app needs access to the camera to take photos.</string>
<key>NSPhotoLibraryUsageDescription</key>
<string>This app would like to access the camera to take a picture of your
pet.</string>
```

NOTE After updating the keys in the Info.plist file, you will need to rebuild your app and restart the run CLI command if you are using it.

Appendix

A

Android emulator tips

If you have previously done mobile development on Android, you may be familiar with some of the common annoyances of the Android emulator such as getting the emulator set up to run and the subpar performance of the Android emulator. This appendix provides some general tips for working with the Android emulator.

A.1 Emulator speed

The Android emulator is known for being notoriously slow (especially on Windows). One way to improve the speed of the Android emulator is to install the Intel Hardware Accelerated Execution Manager (Intel HAXM). By installing Intel HAXM the Android emulator can execute the application code directly on the CPU of your development machine instead of having to first translate the code. Normally, Intel HAXM is installed as part of Android Studio; however, because we are developing in NativeScript, we never had to install Android Studio. To install Intel HAXM, follow the instructions at <https://software.intel.com/en-us/android/articles/intel-hardware-accelerated-execution-manager>

A.2 Using Genymotion

Genymotion is an Android emulator alternative that you can use instead the official Android emulator. Genymotion is free for personal use and has different for-pay licensing schemes for a larger team setting. You can download and install Genymotion from <https://www.genymotion.com/fun-zone/> (you will be required to create an account to get the free personal use license).

B

NativeScript CLI quick reference

The official NativeScript command line interface (CLI) reference is online at <https://www.npmjs.com/package/nativescript>. But, if you're looking for a quick reference, read on! You may notice some commands we left out of the book. That's because they're commands we don't use every day.

B.1 Creating apps

When you need to create a new NativeScript app, you'll start with the `tns create` command. Table B.1 details the various options.

Table B.1 NativeScript CLI commands used when creating apps

CLI Command	Description
<code>tns create <app-name></code>	Creates a new cross-platform NativeScript app named <code><app-name></code> . A folder will be created with the name of your app, and the NativeScript app structure described in chapter 3 will be added. This command also creates a vanilla NativeScript app, using JavaScript.
<code>tns create <app-name> --template typescript</code>	Creates a new vanilla NativeScript app using TypeScript (instead of JavaScript).
<code>tns create <app-name> --template angular</code>	Creates a new NativeScript-with-Angular app using TypeScript and Angular.
<code>tns create <app-name> --tsc</code>	Shortened form of <code>--template typescript</code> .
<code>tns create <app-name> --ng</code>	Shortened form of <code>--template angular</code> .

```
tns create <app-name>
--template <local-or-remote-path>
```

--template can also be used to reference a local or remote template. Include a file system directory or a Github .git URL.

```
tns create <app-name>
--copy-from <directory>
```

Creates a new project from an existing project in the directory specified. This is a great way to copy an existing project, while giving it a new name.

B.2 Adding the Android and iOS platforms

A NativeScript app doesn't do much without adding the Android or iOS platform. Use these commands to add a platform. If you need a reminder of how the `tns platform` command affects your NativeScript app files and folders, check out chapter 3. Table B.2 detailed the various platform commands available in the CLI.

Table B.2 NativeScript CLI commands used when adding native platforms

CLI Command	Description
<code>tns platform add android</code>	Adds the Android platform files to your app
<code>tns platform add ios</code>	Adds the iOS platform files to your app
<code>tns platform remove android</code>	Removes the Android platform files from your app
<code>tns platform remove ios</code>	Removes the iOS platform files from your app

TIP Don't be afraid to remove the Android or iOS platform files from your app. Removing these files doesn't affect the source code of your app. If you think your app isn't running correctly, remove your platforms and add them back. It only takes a few seconds.

B.3 Building apps

Chapters 12 and 13 describe the two-phase process for transforming your app's source code into native Android and iOS projects: the prepare phase and the build phase. These phases use the `tns prepare` and `tns build` CLI commands, detailed in table B.3.

Table B.3 NativeScript CLI commands used when preparing and building apps

CLI Command	Description
<code>tns prepare android</code>	Copies Android-specific settings from the App_Resources folder (and your app's source code) into the native Android platform folder. If the app doesn't have the Android platform installed, the CLI will automatically run <code>tns platform add android</code> .

tns prepare ios	Copies iOS-specific settings from the App_Resources folder (and your app's source code) into the native iOS platform folder. If the app doesn't have the Android platform installed, the CLI will automatically run <code>tns platform add ios</code> .
tns build android	Invokes the Android SDK to compile the files in the Android platform folder into an Android app. This creates a debug version of the app.
tns build android --release --key-store-path <key-store-path> --key-store-password <key-store- password> --key-store-alias <key-store-alias> --key-store-alias-password <key- store-alias-password>	Builds a release version of an Android app. Apps submitted to the Google Play store must be compiled with the --release flag. As outlined in chapter 12, when an app is compiled in release mode, additional key store parameters are required to digitally sign the app.
tns build ios	Invokes the iOS SDK to compile the files in the iOS platform folder into an iOS app. If no iOS device is attached to your computer, the app is built for an emulator. Otherwise it is built for a device.
tns build ios --emulator	Compiles an iOS app for an emulator. Use this when an iOS device is attached to your computer and you want to force the compilation for an emulator.

B.4 Deploying apps

You can use Android emulators, iOS simulators, and real devices to test your app during development. Use the CLI commands detailed in table B.4 during your testing process.

Table B.4 NativeScript CLI commands used when deploying apps to devices, emulators, and simulators

CLI Command	Description
tns device	Displays a list of connected devices to your computer. Physical devices and Android emulators are enumerated by this command. Each device has a device number assigned by the CLI.
tns deploy android	Deploys the app to the first Android device detected by <code>tns device</code> . If your app hasn't been built, the CLI will first invoke <code>tns build android</code> to build your app.

```
tns deploy android  
--device #
```

Specifies a specific Android device to deploy your app to. Useful if you have an emulator and/or multiple physical devices attached.

```
tns deploy ios
```

Deploys the app to the first iOS device detected by `tns device`. If no physical devices are detected, the app is deployed to the iOS simulator. If your app hasn't been built, the CLI will first invoke `tns build ios` to build your app. See our notes below about deploying to physical iOS devices!

```
tns deploy ios  
--device #
```

Specifies a specific iOS device to deploy your app to. Useful if you have a simulator and/or multiple physical devices attached.

```
tns run android
```

Shorthand for `tns prepare android`, `tns build android`, and `tns deploy android`.

```
tns run android  
--device #
```

Just like `tns run android`, but for a specific device.

```
tns run ios
```

Shorthand for `tns prepare ios`, `tns build ios`, and `tns deploy ios`.

```
tns run ios  
--device #
```

Just like `tns run ios`, but for a specific device.

WARNING If you're trying to deploy an iOS app to a physical device, you need to register your device in the iOS Developer Center and sign your app with a digital signature. Chapters 13 and 14 describe this process in detail. If you want the abridged version, check out this brief blog post on iOS code signing: <http://seventhsoulmountain.kripajay.com/2013/09/ios-code-sign-in-complete-walkthrough.html>.

TIP We've mentioned removing the Android and iOS platforms previously, but it's worth pointing out that a combination of `tns remove platform android|ios` and `tns run android|ios` is a shortcut for fully-resetting your app's native platform code and testing it on a device or emulator. The `tns run` command will add the necessary platform, prepare, build, and deploy it automatically. This isn't something you'll have to do regularly, but keep it in your back pocket.

C

NativeScript conventions

Conventions exist everywhere in the real world, and make our lives easier by reducing the number of decisions you need to make daily. Take learning to drive a car as an example – conventions (or rules of the road) are everywhere: drive in the right-hand lane, red lights mean stop, a car with a left blinking tail light means it's going to turn left, and flashing lights or signs generally mean caution – watch out! Without these conventions, you could drive, but it may not be nearly as safe or productive as you'd like.

Just like the rules of the road make driving easier, software development conventions make being a developer easier. But, understanding conventions is important for reasons other than making development easier. Conventions establish a standard or baseline for how code should be written and organized. When writing software, you should expect others will need to update, add to, or maintain the code you have written. By following agreed-upon standards and conventions, your code is more maintainable (which should be one of your goals as a developer). Often, developers feel their job is to be the fastest developer (with a goal of writing the fewest lines of code), but that's wrong. Instead, you should be focusing on writing descriptive, easy-to-read code. In doing so, you allow other developers to better understand the purpose of your code, thus making it more maintainable.

C.1 Understanding NativeScript conventions

NativeScript comes with its own conventions, which make developing mobile apps simpler, more efficient, and less demanding on you. But there's more because these conventions aren't optional – if you don't follow NativeScript's conventions, your apps won't run. As a result, it's important for you to understand the conventions imposed on you by the NativeScript runtime. The faster you learn the conventions, the faster you'll be able to create your second, third, and fourth app. As you learn these conventions, you may find yourself referring to this book or the official NativeScript documentation quite a bit. But with each app you write, you'll become more efficient and these conventions will become second nature.

The NativeScript conventions are mixture of file-naming conventions (to organize your app's user interface and business logic), platform-specific conventions (enabling you to target bits of code and user interface components for a specific mobile platform), and folder structure conventions (to separate

Android and iOS configuration files). You've already seen many of these conventions in this chapter, but let's take a closer look at each.

C.1.1 File-naming conventions and pages

In the first part of chapter 3, you learned how similarly-named files (such as *main-page.xml* and *main-page.js*) come together to form a cohesive unit, called a page.

TIP When you're developing and debugging your NativeScript app, it can be easy to misspell file names. One of the first things I do when an app isn't running as expected is to ensure each page is named properly and has a *<page-name>.xml* and *<page-name>.js* file (and *<page-name>.css*, if you have provided page-specific CSS styles).

As a reminder, when your NativeScript app runs and is told to load a page named *main-page*, the NativeScript runtime knows (by convention) to search for three files: an XML file, a CSS file, and a JavaScript file named *main-page*. The following code snippet from the *app.js* file and figure C.1 show how the NativeScript runtime uses this file-naming convention to start your app.

```
application.start({ moduleName: "main-page" });
```

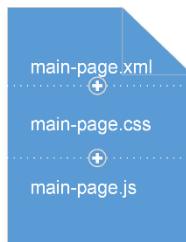


Figure C.1 When the *main-page* page is referenced, the NativeScript runtime looks for and loads files named *main-page.xml*, *main-page.css*, and *main-page.js*.

This convention is powerful, and makes your code easier to read and write. For example, in HTML applications, every HTML file contains tags to include external JavaScript and CSS files. Have you ever thought how cumbersome this is? NativeScript makes your life easier. If you want to load JavaScript and CSS files with your user interface XML files, just name them the same.

Not every page you create will have an XML, CSS, and JS file. At minimum, you will have an XML user interface page, but you may not need to define any page-specific styles or business logic. Something nice about this convention is that you don't need to worry about missing CSS or JS files. When you reference a page by its name (*main-page*), the NativeScript runtime searches for each file (XML, CSS, and JS). If one of the files doesn't exist, it skips the file and continues, loading as many of the files as it can.

This convention is easy to understand and implement, plus it creates a more readable app, as shown in figure C.2.

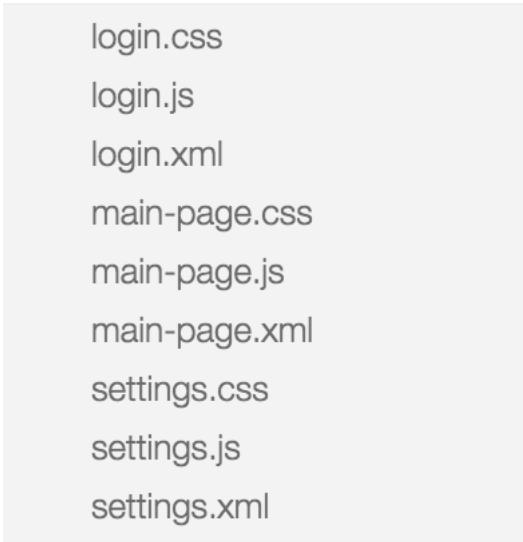


Figure C.2 Similarly named XML, CSS, and JavaScript files make it easy to see there are three distinct pages in this NativeScript.

Because of the similarly named files, it's easy to see there are three pages, each with three related files.

C.1.2 Platform-specific conventions

Although a goal of using NativeScript is to write cross-platform apps with the same code base, this doesn't mean an Android app and an iOS app will share 100% of the user interface and business logic code. In fact, you may find circumstances where you may want to have different user interfaces or different business logic rules, depending on your platform.

NOTE One example for not sharing code is to take advantage of a platform-specific hardware function.

For example, consider iOS's Touch ID. Touch ID is a fingerprint reader that used to verify a person's identity. Some Android devices (like the Google Pixel) have integrated fingerprint readers also, but iOS's Touch ID and Android fingerprint readers are inherently different. If your app needed a user to validate their identity, iOS-specific code may leverage Touch ID, whereas Android code may use an integrated fingerprint reader (if present) or fall back to a username and password.

NativeScript provides several mechanisms for you to choose from if you're trying to separate your user interface or business logic code by platform. Let's look at each.

SPLITTING XML FILES TO VARY BY PLATFORM

The first way to provide platform-specific user interface code is to split your XML file into two separate files. Once split, you will place Android-specific user interface code in one file, and iOS-specific code in a

second file. To differentiate the two files, you use another file-naming convention, by adding `.android` or `.ios` to the file name.

Let's assume you are starting with a platform-agnostic page currently displaying a button with the following XML code: `<Button text="Tap Here!" />`. Figure C.3 shows you the user interface code and resulting platform rendering on Android and iOS of the button.



Figure C.3 Platform agnostic user interface markup within a single file will display the same button on both Android and iOS.

You'll notice the `main-page.xml` file is the only file included in the app; therefore, both Android and iOS platforms will share the user interface code.

Now, imagine you would like to customize the button's text based on the platform. Figure C.4 shows how this platform-specific file-naming convention can be used to split the main-page example you saw earlier in this chapter.

main-page.android.xml
main-page.ios.xml

Figure C.4 The `main-page.xml` file has been split into two separate XML files: `main-page.android.xml` and `main-page.ios.xml`.

Using the `.android` and `.ios` file-naming convention, the original `main-page.xml` file has been duplicated, with the original and duplicate renamed to `main-page.android.xml` and `main-page.ios.xml`.

```
<Button text="Tap Here for Android! " />
<Button text="Tap Here for iOS! " />
```

The first line of XML code is placed in the Android-specific file, and the second line is placed in the iOS-specific file. Figure C.5 show the results of splitting the `main-page.xml` file into platform-specific files and changing the button text accordingly.

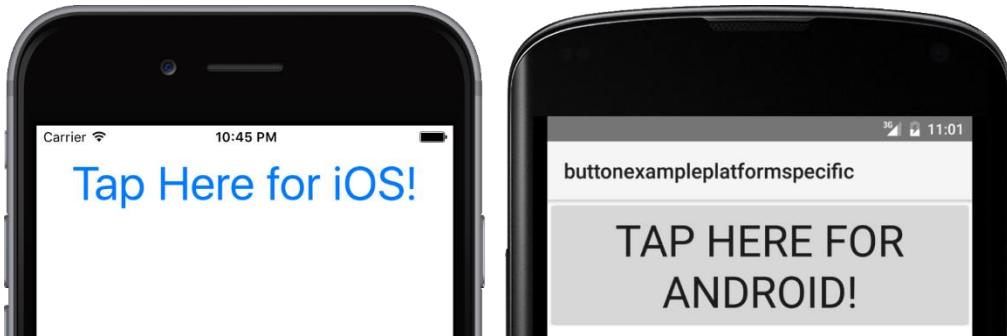


Figure C.5 By using the platform-specific file-naming convention, the Android and iOS apps have different button text.

When you use the platform-specific file-naming convention of `.android` and `.ios`, NativeScript automatically associates the correct user interface code with the appropriate platform. The Android app will load the `main-page.android.xml` file, displaying the Android-specific button text, and the iOS app will load the `main-page.ios.xml` file, displaying the iOS-specific button text.

As you can see, it's easy to create platform-specific user interface views: create two XML files with the `.android` and `.ios` file-naming convention. Although this is easy, this approach requires you to duplicate the user interface for each platform. Imagine a complex page with images, buttons, and text. Maintaining two separate pages just so you can change the text on a button seems cumbersome. So, using this approach for supporting platform-specific user interface customizations only makes sense when you're changing a large portion of the user interface.

SHARING XML FILES TO VARY BY PLATFORM

If you have a relatively small number of user interface customizations, NativeScript offers a simple way of customizing the user interface. Instead of splitting your XML file into two files, you maintain a single XML file, but add additional XML markup to the file to identify platform-specific customizations. This approach is shown in Listing C.1.

Listing C.1 Platform-specific user interface customizations using XML markup

```
<android>                                // #A
  <Button text="Tap Here for Android!" />  // #A
</android>                                 // #A
<ios>                                     // #B
  <Button text="Tap Here for iOS!" />    // #B
</ios>                                      // #B
#A By using the <android> markup, this button will only be displayed on devices running Android
#B User interface markup within an <ios> tag will only be displayed on devices running iOS
```

By using the platform-specific XML markup `<android>` and `<ios>` within your app, NativeScript will selectively display user interface code within each tag on the appropriate platform. Figure C.6 shows the results of using the `<android>` and `<ios>` tags.

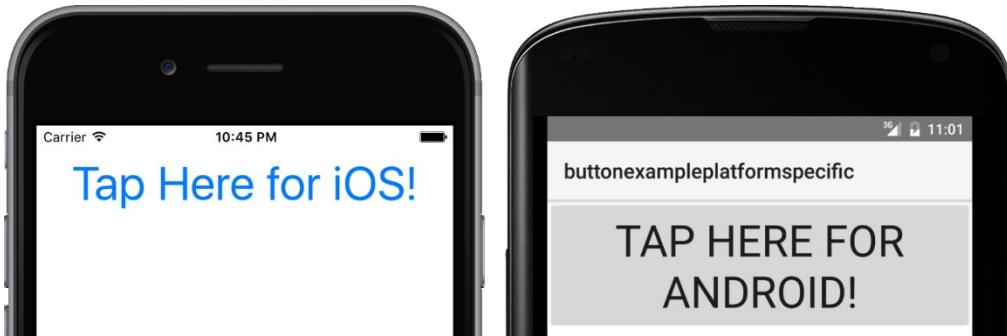


Figure C.6 Using platform-specific `<android>` and `<iOS>` tags, the Android device displays Android-specific code, and the iOS device displays iOS-specific code.

SEPARATING JAVASCRIPT FILES TO VARY BY PLATFORM

Earlier in this chapter, you learned how to create platform-specific user interface pages by using the `.android` and `.ios` file name convention. The same concept applies to your business logic code written in JavaScript.

Let's assume you want to create a button via JavaScript code and set the text of the button based on the platform, as shown in listing C.2.

Listing C.2 Creating a button via JavaScript

```
var buttonModule = require("ui/button"); //#A
var button = new buttonModule.Button(); //#B
button.text = "Tap Here!"; //#B
#A import the button module, saving a reference in the buttonModule variable
#B create a native button using JavaScript and assigning text
```

Right now, it's still not important that you understand everything happening in this code snippet, because I will cover everything in future chapters. What is important is that you understand that it's possible to create a button and specify the text property via code.

Consider the same page described earlier in this chapter (*main-page*). If the code in listing C.2 exists within `main-page.js`, NativeScript produces identical buttons that are platform agnostic. You can see this in figure C.7.



Figure C.7 Platform agnostic JavaScript code within a single file will display the same button on both Android and iOS.

To create platform-specific JavaScript code, the *main-page.js* file needs split into two files with a *.android* and *.ios* file-naming convention applied. Listing C.3 and C.4 illustrate the code differences between the two files.

Listing C.3 Platform-specific code customizations using JavaScript from main-page.android.js

```
var buttonModule = require("ui/button");
var button = new buttonModule.Button();
button.text = "Tap Here for Android!";
```

Listing C.4 Platform-specific code customizations using JavaScript from main-page.ios.js

```
var buttonModule = require("ui/button");
var button = new buttonModule.Button();
button.text = "Tap Here for iOS!";
```

You'll notice the Android and iOS code looks almost identical, except for the value assigned to the text property of the button. Following the *.android* and *.ios* file-naming convention, the first code portion is placed into *main-page.android.js*, and the second code portion is placed into *main-page.ios.js*. When the code runs on each platform, you get customized buttons (figure C.8).



Figure C.8 By using the platform-specific file-naming convention on JavaScript files, the Android and iOS apps have different button text.

Using this approach to write platform-specific business logic code has the same shortcomings as splitting our user interface code into separate files: code duplication and decreased maintainability. Truthfully, it's painful to use the file-naming convention and duplicate much of your code. Wouldn't it be nice if there were some way to share code between platform-specific JavaScript files? Luckily, there is!

An additional file-naming convention NativeScript uses for JavaScript files is the *.common* convention. This convention works in conjunction with JavaScript files using the *.android* and *.ios* convention. I like to think of the *.common* convention like base or foundational code – when NativeScript interprets your app and comes across JavaScript files adhering to the *.common*, *.android*, and *.ios* file-naming convention, the *.common* JavaScript file is loaded first, creating a base (or foundation), followed by a secondary load

of the platform-specific `.android` or `.ios` file. Figure C.9 shows how the combination of `.common`, `.android`, and `.ios`.

The diagram consists of a light gray rectangular box containing three lines of text, each representing a separate file:

- main-page.android.js
- main-page.common.js
- main-page.ios.js

Figure C.9 The platform-agnostic `main-page.js` file is split into 3 files: `main-page.common.js`, `main-page.android.js`, and `main-page.ios.js`. On Android, `main-page.common.js` and `main-page.android.js` are combined. On iOS, `main-page.common.js` and `main-page.ios.js` are combined.

To use the `.common` file-naming convention with a platform-agnostic `main-page.js` file, you split the file into three files named `main-page.common.js`, `main-page.android.js`, and `main-page.ios.js`. When your app runs on Android, the `main-page.common.js` file is combined with the `main-page.android.js` file, and `main-page.common.js` is combined with `main-page.ios.js` on iOS.

Listing C.5 shows an example of how you may structure your JavaScript code to take advantage of the `.common` file-naming convention.

Listing C.5 How to use the `.common` file-naming convention to create platform-specific JavaScript code and reduce the amount of duplicate code written

```
var buttonModule = require("ui/button"); //#A
var button = new buttonModule.Button(); //#A
button.text = GetButtonText(); //#A

function GetButtonText() { //#B
    return "Tap Here for Android!"; //#B
} //#B

function GetButtonText() { //#C
    return "Tap Here for iOS!"; //#C
} //#C

#A Base (or foundational) code is placed in main.common.js. Note how the button text is now set via a function.
This function will not be defined in main-page.common.js, but instead in the platform-specific files
#B Found in the main-page.android.js file, this Android-specific code returns the button text for Android devices
#C Found in the main-page.ios.js file, this code return platform-specific button text for iOS
```

When using the `.common` file-naming convention your code will need to be changed slightly to abstract or pull-out code that should be placed into the `.common` file. Listing C.5 shows how `main-page.common.js` has the shared base code to create a button and set the text of the button. You will notice the button's text will now be set by a function that is not present within the `main-page.common.js` file. Instead, the `GetButtonText()` function is defined and implemented in `main-page.android.js` and `main-page.ios.js`. Each implementation is then customized to accommodate its specific platform.

Now that you've seen how to use the `.common` file-naming convention to reduce the amount of duplicate code written for platform-specific JavaScript files, you see how splitting code between files can be easy and relatively efficient.

SHARING JAVASCRIPT FILES TO VARY BY PLATFORM

Although splitting platform-specific JavaScript code into `.common`, `.android`, and `.ios` files is easy to do, there's an even easier method for writing platform-specific JavaScript code. You don't have to split your JavaScript files to write platform-specific code. Instead, you can use global JavaScript variables to dynamically determine whether your app is running on Android or iOS. Listing C.6 shows how to use this approach by creating a button and setting the text based on the platform.

Listing C.6 Using the android and ios global variables to dynamically determine the device platform and write platform-specific code

```
var buttonModule = require("ui/button"); //#A
var button = new buttonModule.Button(); //#A

if (android) { //#B
    button.text = "Tap Here for Android!"; //#B
}

if (ios) { //#C
    button.text = "Tap Here for iOS!"; //#C
} //#C

//A Similar code is used to create a button
//B The global variable android is used. If android is not NULL, the button text is set to an Android-specific value
//C The global variable ios is used. If ios is not NULL, the button text is set to an iOS-specific value
```

When NativeScript runs on Android and iOS, global JavaScript variables named `android` and `ios`, respectively, are injected into the JavaScript virtual machine running within your app. To check the platform, a simple `if (android)` and `if (ios)` statement can be used. In listing C.6, I use this approach to set the button's text property after detecting which platform the app is running on.

Using the global `android` and `ios` variables is a quick way to add in short, concise platform-specific code sections. I like it because it gives you a lot of flexibility without the hassle of splitting up your JavaScript file into separate `.android` and `.ios` files.

The convenience of using the global variables can be quickly overused, however. If you have large amounts of platform-specific code sections, your code may be difficult to read and understand. I don't feel there's specific guidance on when you should switch from using the global variables to splitting files. But, if you have a couple of global variables references in your code, you're probably ok to stick with that approach. On the other hand, if you have a dozen uses of global variable references, you may want to consider splitting your code into separate files.

What about that gray area when you have 4 to 8 global variable references? That's up to you; if you've structured your code well and it's very readable, stick with the global variables. Otherwise, split it out. Follow your instincts, and if you're in doubt, ask someone else for their opinion.

SEPARATING USER INTERFACE CSS FILES TO VARY BY PLATFORM

As you have seen with user interface (XML) and business logic (JavaScript) files, you can also use the `.android` and `.ios` file-naming convention to write platform-specific user interface styles. In practice,

separating a CSS file into two platform-specific CSS files with a slightly different name works exactly like it works with XML files, so I'm not going into detail. Instead, I want to call out that it is possible to create platform-specific CSS files by naming your CSS files with *.android* and *.ios*.

C.1.3 Screen size conventions

Similar to the *.android* and *.ios* file-naming conventions discussed earlier in this chapter, NativeScript also provides support to tailor which files will be loaded based on screen size by changing the name of a file. The screen size file-naming convention allows you to easily create different user interfaces to target multiple device sizes.

While you're learning about this convention, I'm going to introduce you to a concept called device-independent pixels (dp for short). Device-independent pixels can be a complex topic and work in subtly different ways across Android and iOS. Although it's not important that you understand the intimate details of device-independent pixels, you should know that it's a way of describing the physical size (width and height) of a mobile device's screen, without describing the number of pixels.

TIP Understanding device independent pixels (dp) and their relating concepts of dots, points, DPI, and display scaling can be confusing on mobile devices because Android and iOS handle them differently. Although an in-depth discussion of these concepts is not covered in this book, you'll understand how these concepts relate to each other by reading this blog post: <http://blog.fluidui.com/designing-for-mobile-101-pixels-points-and-resolutions/>.

In general, Android and iOS define and classify their screens to have ~160 dp per inch. The number is different on each platform, but thinking at 160 dp per inch is good enough for now.

Although the lines between phone and tablet are getting harder to differentiate because of larger and larger phone screens, you can use the ~160 dp per inch guideline to tell what form factor your device is. The generally-accepted rule is if a device's smaller dimension (width or height) is more than 600 dp (~3.75 inches), the device is considered a tablet.

As we continue to explore NativeScript's screen size convention, just keep in mind that device-independent pixels are a method for measuring screen size.

To use the screen size convention, you can add one of the following naming conventions to your file:

- *minW#*—Page displayed if the device width is at least # dp
- *minH#*—Page displayed if the device height is at least # dp
- *minWH#*—Page displayed if the smaller of dimensions (width or height) is at least # dp

Assuming I have two devices (a phone and a tablet) and want to display two different versions of *main-page.xml* based on the form factor, I create two versions of the page, named *main-page.minWH600.xml* and *main-page.xml*. *Main-page.minWH600.xml* is displayed on the tablet, and *main-page.xml* is displayed on the phone. Let's look at an example where I have two different buttons displayed using this convention. The first line of code is contained within the *main-page.minWH600.xml* file, and the second within the *main-page.xml* file. Figure C.10 shows the results when run on a mobile device.

```
<Button text="Tablet" />
```

```
<Button text="Phone" />
```



Figure C.10 A phone and tablet displaying different pages based upon the file-naming convention used. NativeScript will display the `main-page.minWH600.xml` file on the tablet, and the `main-page.xml` file on the phone.

When NativeScript loads the `main-page` page, it detects the screen size in device-independent pixels and chooses the appropriate file to display. Tablets are greater than 600 dp, so `main-page.minWH600.xml` is displayed, thus showing a button with the text of "Tablet." Phones, which have a screen size less than 600 dp, will display `main-page.xml`, and a button with the text of "Phone."

C.1.4 Screen orientation conventions

Another common need when developing apps for mobile devices is to vary the user interface based upon the screen orientation. With NativeScript, you can use the `.port` and `.land` file-naming conventions to account for holding your device in a portrait or landscape position. Figure C.11 shows how this is done by placing `<Button text="Portrait" />` in the `main-page.port.xml` file, and `<Button text="Landscape" />` in the `main-page.land.xml` file.

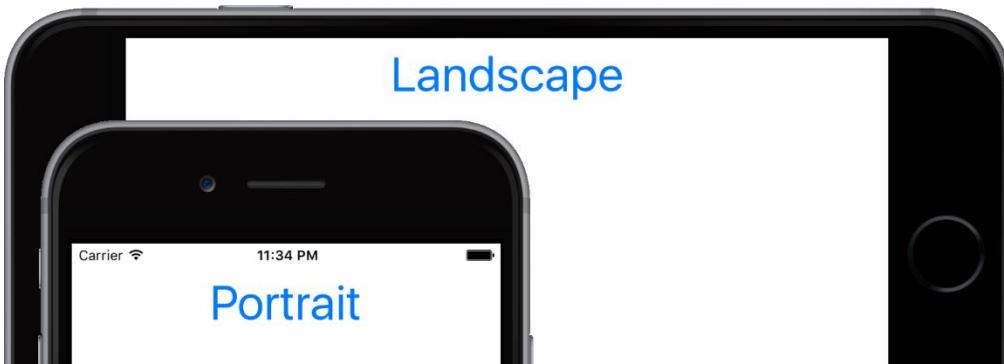


Figure C.11 A phone in a portrait and landscape position. The portrait position displays the `main-page.port.xml` file and the landscape position will display the `main-page.land.xml` file.

This file-naming convention works just like the other conventions you learned about in this chapter. By naming your page file *main-page.port.xml*, NativeScript displays the page when the mobile device is placed in a portrait position. Devices placed in a landscape position will display *main-page.land.xml*.

17.8.1 Chaining conventions

You've just learned about several file-naming conventions that can be used to control the screen orientation, screen size, and targeted platform. But, in all our examples, we only applied one convention at a time. What if you need to apply multiple conventions at once?

TIP File-naming conventions can be chained (or concatenated) together to target a complex combination of traits.

For example, to create a landscape page for Android tablets, name the page *main-page.land.minWH600.android.xml*.

D

Creating custom UI controls

In chapter 11, you learned how to create more consistent and professional-looking apps by creating a tablet-specific version of the Pet Scrapbook. The tablet-specific pages of the Pet Scrapbook rely on copying and pasting a lot of code. For a small app, copying and pasting some code isn't that bad, but as apps grow larger and share UI and JavaScript code between pages, it's important to centralize the code you write.

In this appendix, we'll take you on a journey of creating reusable UI components that incorporate data binding. We'll continue to use the Pet Scrapbook in our examples. If you'd like to follow-along, go back to the Pet Scrapbook version we had at the end of chapter 11.

NOTE Before you begin, delete the contents of the tablet-specific page files (`scrapbook-page.land.minWH600.xml` and `scrapbook-page.land.minHW600.js`).

D.1 *Introducing reusable, custom UI controls*

Before we jump into creating our first reusable custom UI control, let's slow down. You may be wondering what a reusable UI control is and how it works.

DEFINITION Reusable UI controls are portions of XML and/or JavaScript code that are written once and bundled together. Once bundled together, they act just like the UI elements you've learned about (labels, text fields, buttons, and so on) and can be added to the XML code of pages.

Developing custom UI controls is relatively easy, and because you're creating the control, it can be as simple or as complex as you'd like. For example, a simple UI control could be a label: `<Label text="I am a reusable UI component!" />`. A complex UI control could be a collection of UI elements (like a label, text field, and button) with the ability to dynamically bind to various properties.

Custom UI controls are built in one of two ways:

3. Control is defined via XML code only (a.k.a. simple custom UI control)

4. Control is defined via JavaScript code only (a.k.a. complex custom UI control)

Both methods for creating custom UI controls produce similar results: a write-once-use-multiple places custom UI control. But, there are some limitations to the XML-only approach, namely the ability to define custom properties. For example, you couldn't use the XML-only approach if you wanted to create a custom UI control named *Pet* with a *petName* property like so: <Pet petName="Pkitty" />.

In this section, you'll learn how to create both simple and complex custom UI controls. Let's get started with a simple example.

NOTE In practice, we find ourselves creating complex UI controls with dynamic data-binding support more than the simple versions, and the final version of the Pet Scrapbook will use two complex UI controls. Even though we won't be using simple UI controls, it's important that you learn the basics before jumping into more complex reusable UI controls.

D.1.1 Creating a simple custom UI control

Simple custom UI controls are a collection of one or more UI elements bundled into an XML file.

TIP Before we get started, we need a place to put our custom UI controls. We recommend creating a folder inside of your views named *shared*. Inside the *shared* folder, you'll have an additional folder that contains the JavaScript file for each custom UI control.

Let's create the simplest custom UI control by following these short steps:

1. Create a folder named *simple-label* inside of the *views/shared* folder. The shared folder will help us keep all our custom controls organized and separate from the other pages, and the *simple-label* folder will contain the code files for one of our custom UI controls.
2. Add a new file named *simple-label.xml* to the *simple-label* folder.
3. Add a label to the *simple-label.xml* file: <Label text="I am a reusable UI component!" />.

That's it! You've created your first custom UI control. We said it would be simple. Let's use it by adding it to the tablet-specific scrapbook page (*scrapbook-page.land.minWH600.xml*).

ADDING A CUSTOM UI CONTROL TO A PAGE

To use the simple label control we just created on a page, we'll have to do two things:

1. Add a reference to the control's location to the page.
2. Add an XML element referencing the custom UI control.

This probably feels a bit confusing, so let's walk through it carefully. The first step is to add a reference to the control's location. This is done by adding an *XML namespace declaration* to the page element.

DEFINITION XML namespaces are used as a method for classifying elements within an XML document. Just like namespaces in object-oriented programming languages, XML namespaces allow you to create multiple classes with the same name and use them in the same program. With namespaces, you can have two classes named *Pet*, with each belonging to a different namespace. For example, the *Good*

namespace could have a Pet class, and the Bad namespace could have a Pet class. XML namespaces do the same thing, but for XML elements instead of classes. XML namespaces are key-value pairs and can be declared by using the `xmlns:{namespace-prefix}="namespace-name"` syntax. If you'd like to learn more about XML namespace, check out https://en.wikipedia.org/wiki/XML_namespace.

Although the primary purpose of XML namespace is to classify elements, NativeScript also uses an XML namespace declaration as a reference (or pointer) to an element's code-based implementation. For example, let's look at the XML namespace declaration for our simple label custom UI control: `xmlns:shared="views/shared/simple-label"`. When placed in a page, this declaration tells the NativeScript runtime several things, as described in table D.1.

Table D.1 The simple label XML namespace declaration decomposed

Namespace part	Description
<code>xmlns</code>	Standard XML syntax for declaring a namespace. All namespaces begin with this value.
<code>:shared</code>	A prefix (or alias) for the namespace that is used instead of the often-longer namespace name. The shorter prefix is then used as a XML tag name prefix throughout an XML document. In our example, the prefix is named <code>shared</code> , and is a shortened form of <code>shared/simple-label</code> .
<code>shared/simple-label</code>	The namespace name, also used by NativeScript to identify the relative file path and location of a custom UI control's code file (without the .xml file extension). In our case, this points to the file named <code>simple-label.xml</code> in the <code>shared</code> folder.

Go ahead and add the XML namespace declaration to the tablet-specific scrapbook page by adding it to the `page` element's opening tag: `<Page xmlns:shared="views/shared/simple-label">`.

NOTE The XML namespace prefix is an arbitrary name, and can be any value. We chose the name `shared`, but could have chosen any other value, such as `custom-label` or `sl` (simple label abbreviated).

The second step is to add the custom UI control to the page as an XML element. This is done just like adding other UI elements to the page, but with the addition of the namespace prefix: `<shared:simple-label />`. Listing D.1 shows the code for a page with several instances of the simple label added.

Listing D.1 A page with several instances of the simple label custom UI control added

```
<Page xmlns:shared="views/shared/simple-label">
<StackLayout>
  <shared:simple-label />
  <shared:simple-label />
  <shared:simple-label />
</StackLayout>
```

```
</Page>
```

Because we've referenced the simple label custom UI control with the XML namespace declaration, it can be treated just like any other UI element and added to the page's UI as many times as we'd like. Viewing this page would result in three labels displayed on top of each other (figure D.1).

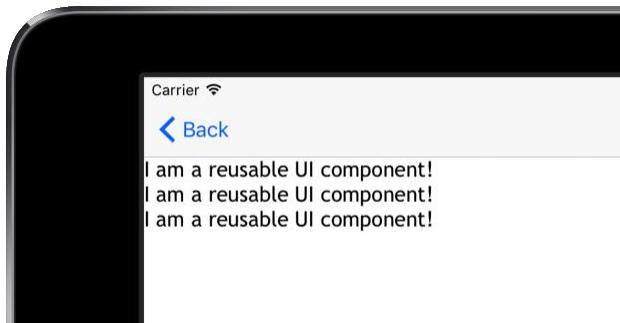


Figure D.1 Three reusable UI controls stacked on top of each other.

This was a brief introduction to simple, XML-only custom UI controls. There's more to learn about the XML-only approach, but we honestly don't use this type of custom UI control often. We don't want to hold you back, so if you'd like to learn more, the NativeScript documentation goes into more detail: <http://docs.nativescript.org/ui/basics#custom-components>.

D.1.2 *Creating complex custom UI controls*

In the last section, you learned how to create a simple custom UI control by adding UI elements to an XML file. It was very similar to building pages, but with a few minor twists. In this section, you're going to learn how to create a custom UI control via JavaScript. As we mentioned earlier, we find ourselves creating custom UI controls with JavaScript more frequently because it provides additional capabilities and flexibility.

TIP Building custom UI controls with JavaScript allows you to integrate custom data binding and custom events, both of which you cannot do if you use the XML-based approach.

Let's get started by creating a custom UI control to replace the list of scrapbook pages that we built in chapter 11, as shown in figure D.2.

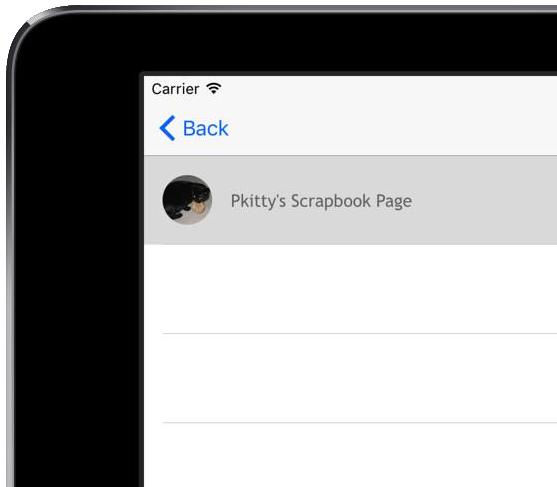


Figure D.2 The tablet-specific page showing a list of scrapbook pages on the left of the screen.

We'll be building the custom control for the list in several stages:

4. Create a new JavaScript object that inherits from an existing layout control.
5. Create a function that builds the control's UI elements via code.
6. Define custom UI control properties with `Object.defineProperty()`.
7. Configure custom data-binding for the control's UI elements.
8. Configure custom events for the control's UI elements.

Wow! There seems to be a lot going on in those five steps, and truthfully, there is, but we'll take it slow and explain each step along the way. When we're finished, you'll be able to use the custom control on any page by adding it to the XML code: `<shared:ScrapbookList items="{{ listItems }}" />`.

WARNING Creating custom UI controls with JavaScript is an advanced topic, using more advanced JavaScript techniques, and a lot of code. If you feel like you're stuck, jump back to our outline of the five steps to building custom UI controls to regain your footing.

STEP 1: CREATE A NEW JAVASCRIPT OBJECT THAT INHERITS FROM AN EXISTING LAYOUT CONTROL

Before we dive in, let's create the file and folder structure to house our custom UI control. We're going to call our custom UI control `ScrapbookList`, so create a folder named `scrapbook-list` underneath the `views/shared` folder. Then add a file named `scrapbook-list.js`.

The first step in creating a custom user control is to create the shell of the control that is based upon a layout control. We'll have to decide because there are several layout control choices to use as a base (stack, grid, wrap, and dock layouts), but the grid layout stands out because it provides a mechanism for consuming the available space on a page.

NOTE You'll recall from chapter 5 that by setting the rows and columns property of a grid layout with percentage-based sizing (rows="*" and columns="*"), you can force the grid row and column to consume the maximum amount of space available.

Let's look at the base custom UI control code in listing D.2.

Listing D.2 Base JavaScript code for a custom UI control

```
var GridLayout = require("ui/layouts/grid-layout").GridLayout; #A

var ScrapbookList = (function (_super) {           #B
    global.__extends(ScrapbookList, _super);   #C

    function ScrapbookList() { #D
        _super.call(this);      #D

        this.rows = "*";       #D
        this.columns = "*";    #D
    }                                #D
    return ScrapbookList;          #D

}) (GridLayout); #E

exports.ScrapbookList = ScrapbookList;
#A Modified import statement to only pull the Grid Layout object out of the grid-layout module
#B Custom controls are structured as self-executing functions; _super is the inherited control (GridLayout)
#C Performs the actual inheritance of the Grid Layout object
#D Function behaves like a constructor, initializing a new Grid Layout
#E The Grid Layout object is passed into the self-executing function and becomes _super
```

Wow's that a lot of new code that you may not be familiar with, especially if you haven't delved into advanced JavaScript development techniques. So, let's break it down.

Starting at the top, you should recognize the `require("ui/layouts/grid-layout")` statement, with the exception of the `.GridLayout` command added to the end. The syntax may look confusing, but it's a shortcut for obtaining a reference directly to the grid layout module's Grid Layout object. In other words, it's equivalent to the following:

```
var gridLayoutModule = require("ui/layouts/grid-layout");
var GridLayout = gridLayoutModule.GridLayout;
```

Moving to the next large code block you see the definition of our custom UI control. Custom UI controls are built as immediately invoked function expressions (IIFE) with a single argument.

DEFINITION Immediately invoked function expressions (IIFE) are a JavaScript construct that executes (or invokes) the function immediately. Because of the way the function is written and executed, the contents of the function receive a sand-boxed, local function scope that is private and cannot be accessed externally. You can read more about IIFEs at https://en.wikipedia.org/wiki/Immediately-invoked_function_expression.

The IIFE ends with a reference to the Grid Layout object being passed in as an argument. When passed in, the Grid Layout object is known as `_super`, and will eventually be used as the object we inherit from.

Next up in listing D.2 is the call to `global.__extends(ScrapbookList, _super);` inside of the IIFE. This function call creates the inherited relationship between the `ScrapbookList` object and `_super` (a.k.a. the Grid Layout object of the grid layout module).

We find the best way to think of the `ScrapbookList()` function that follows the inheritance statement is as a constructor: when someone creates a new instance of our control, two things happen:

1. a new grid layout object is created with the `_super.call(this);` command
2. the rows and columns properties are set to `*` (1 row, 1 column, both taking the maximum space available)

That's all for the base of a custom UI control.

TIP If you're filing away code snippets for reuse in your own apps, listing D.2 is one to keep. You can use this base code as a template for all your custom UI controls, swapping out the grid layout for another layout, when appropriate.

STEP 2: CREATE A FUNCTION THAT BUILDS THE CONTROL'S UI ELEMENTS VIA CODE

At this point, our base custom UI control doesn't display anything on the screen except an empty grid layout (that's because the custom UI control is just a grid layout). In this step, we'll be adding controls to the grid layout.

Let's look at the scrapbook list XML code we built in chapter 11 so we have a reference to the XML elements we need to add to the grid layout (listing D.3).

Listing D.3 The scrapbook-page.xml page showing the list view XML code

```
<Page loaded="onLoaded">
  <Page.actionBar>...</Page.actionBar>

  <GridLayout rows="*" columns="*"
    items="{{ pages }}"
    itemTap="onItemTap">
    <ListView.itemTemplate>
      <StackLayout orientation="horizontal" class="{{ isActive ?? 'list-group-item active' : 'list-group-item'}}">
        <Image class="thumb img-circle" src="{{ image }}"/>
        <Label class="list-group-item-text" style="width: 100%; textWrap="true"
          text="{{ title, (title === null || title === undefined ? 'New' : title + '\\'s') + ' Scrapbook Page' }}"/>
      </StackLayout>
    </ListView.itemTemplate>
  </ListView>
</GridLayout>
</Page>
```

#A Content to be added to our custom UI control via JavaScript

The grid layout from the `scrapbook-page.xml` file contains a single element: a list view. But, how are we going to add this XML to the custom UI control? The custom UI control is in JavaScript. We'll call your attention back to chapter 3, where you learned that each UI element is a JavaScript object and can be

created by using JavaScript code. We'll be using this feature to add the list view to our custom UI control, as shown in listing D.4.

Listing D.4 The scrapbook list custom UI control with the list view UI element added to the grid layout via JavaScript

```
var GridLayout = require("ui/layouts/grid-layout").GridLayout;
var ListView = require("ui/list-view").ListView;
var StackLayout = require("ui/layouts/stack-layout").StackLayout;
var Image = require("ui/image").Image;
var Label = require("ui/label").Label;

var ScrapbookList = (function (_super) {
    global.__extends(ScrapbookList, _super);

    function ScrapbookList() {
        _super.call(this);

        this.rows = "*";
        this.columns = "*";

        var listView = new ListView(); #A
        listView.className = "list-group";

        listView.itemTemplate = function() { #B
            var stackLayout = new StackLayout();
            stackLayout.orientation = "horizontal";
            stackLayout.bind({ #C
                targetProperty: "className",
                sourceProperty: "$value",
                expression: "isActive ?
                    'list-group-item active' : 'list-group-item'" #C
            });

            var image = new Image();
            image.className = "thumb img-circle";
            image.bind({ #C
                targetProperty: "src",
                sourceProperty: "image"
            });
            stackLayout.addChild(image);

            var label = new Label();
            label.className = "list-group-item-text";
            label.style.width = "100%";
            label.textWrap = true;

            label.bind({ #C
                targetProperty: "text",
                sourceProperty: "title",
                expression: "(title === null || title === undefined ? #C
                    'New' : title + '\\\'s') + ' Scrapbook Page'" #C
            });
            stackLayout.addChild(label);
        };
        return stackLayout;
    }
});
```

```

    };
    this.addChild(listView); #D
}
return ScrapbookList;
}) (GridLayout);

exports.ScrapbookList = ScrapbookList;
#A The list view is created in the constructor
#B Item templates are created by setting the property to a function that returns a UI element
#C To data bind properties in the item template, you use the bind() function
#D The configured list view gets added as a child to "this" object (which is a grid layout)

```

At the very top of listing D.4, we added a reference to several of the core modules that we'll be using in code: the list view, stack layout, image, and label.

TIP If you're not sure which core module contains the UI element you're looking for, you can search through your app's `node_modules/tns-core-modules` directory to find it, or you can consult the NativeScript API documentation at <http://docs.nativescript.org/api-reference/>. If you use the API documentation, you can perform a quick search of the page in your browser.

The first task is to create the list view UI element: `<ListView class="list-group" items="{} pages {}" itemTap="onItemTap">`. We do this by creating a list view object and setting the `cssClass` property:

```

var listView = new ListView();
listView.cssClass = "list-group";

```

You may have noticed the `items` and `itemTap` properties, but there's no code. That's ok, and on purpose. Don't worry, we'll circle back to these properties once we've finished adding the basics for the list view.

Next, we add the code for the list view's item template. In the XML code, the item template consisted of a stack layout (with a horizontal orientation), an image, and a label, which together look like figure D.3 when displayed on the screen.

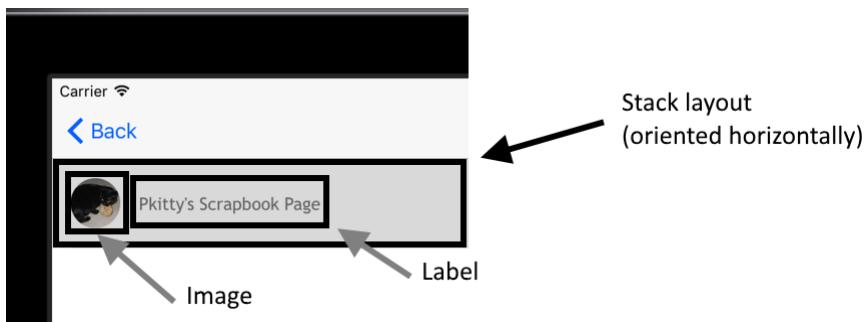


Figure D.3 The list view item template showing a stack layout (oriented horizontally), an image, and a label.

To define a list view item template, we set the `itemTemplate` property to a function that returns the stack layout, image, and label. Defining the properties of these UI elements is straightforward. For example, instead of setting the `orientation` property of the stack layout in XML (`orientation="horizontal"`), the `orientation` property of the stack layout object is set: `stackLayout.orientation = "horizontal"`.

The process of property mapping from XML to JavaScript applies to most of the properties, except for properties using the data-binding mustache syntax. In XML code, the mustache syntax was an easy way to configure data binding; however, when you're defining UI elements, you can't use mustache syntax.

TIP To convert data binding mustache syntax to JavaScript, you must manually bind a property with the `bind()` function.

Back in chapter 8, you learned about mustache syntax and the `bind()` function (which is the manual version of mustache syntax used in JavaScript). We use the `bind` syntax to manually bind the stack layout's CSS class name property, the image's source property, and the label's text property in listing D.4. Let's take a closer look at the label's text property data binding code (listing D.5)

Listing D.5 A closer look at the list view item template's label data binding code

```
label.bind({
    targetProperty: "text",
    sourceProperty: "title",
    expression: "(title === null || title === undefined ?
        'New' : title + '\\\'s') + ' Scrapbook Page'"
});
```

As you'll recall from chapter 8, the `bind()` function takes two parameters (a binding options object, and a reference to the observable we will data bind to). The binding options object has several properties:

- The *target property* identifies which property of the label we want to dynamically change with data binding.
- The *source property* is the property on our data bound observable object.
- The *expression* describes the value to be displayed in the target property.

WARNING You may notice that the `bind()` function is missing a parameter: the data-bound observable. It's not a mistake and is not required in this instance. It may seem confusing, but because the label exists within the list view's item template, the observable object will automatically be set and assigned to the label's binding context when displayed.

Phew! That was a lot of new code, but our custom UI control is starting to come together. We've created the control base (which inherits from a grid layout) and added the control's contents to the grid layout via JavaScript. But we skipped over several properties on the list view: the `items` property (which supplies the data bound observable array of list items), and the `itemTap` property (which identifies an event handler for when a list item is tapped):

```
<ListView class="list-group">
```

```
    items="{{ pages }}" itemTap="onItemTap">
```

In the final three steps, we'll work to add these items to our custom UI control. Let's get started with the `items` property.

STEP 3: DEFINE CUSTOM UI CONTROL PROPERTIES WITH `OBJECT.DEFINEPROPERTY()`

Our primary reason for creating a custom UI control with JavaScript instead of XML was because we wanted to have a custom property named `items` that could provide data binding for the internal UI elements. With this in mind, once we're done, we'll be able to add the custom UI control to any page with the following XML: `<shared:ScrapbookList items="{{ listItems }}" />`.

To add custom properties (like the `items` property) to a custom UI control, you need to use a special JavaScript function named `Object.defineProperty()`.

DEFINITION The `Object.defineProperty()` function is a standard JavaScript function that defines a new property directly on an object. The function requires two parameters: the object on which to define the property and the name of the property to define. We won't be going into detail on how this function adds the property, but you can dive deeper if you're interested by reading https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty.

Listing D.6 outlines how to use this function to create the `items` property.

Listing D.6 Using the `Object.defineProperty()` function to create an `items` property on the scrapbook list custom UI control

```
var ScrapbookList = (function (_super) {
  global.__extends(ScrapbookList, _super);

  Object.defineProperty(ScrapbookList.prototype, "items", { #A
    get: function(){          #B
      return this._items;     #B
    },
    set: function(value){    #B
      this._items = value;   #B
      this.bindData();       #B
    }
  });
}

function ScrapbookList() {
  _super.call(this);

  this._items; #C

  this.rows = "*";
  this.columns = "*";

  ... #D
}

return ScrapbookList;

}) (GridLayout);
```

```
exports.ScrapbookList = ScrapbookList;
#A Properties are defined with the Object.defineProperty() function, this property is named "items"
#B All properties have a get() and set() function used to access the private, internal backing object
#C A private, internal backing object is required for all properties
#D Additional code from constructor left out intentionally
```

Although the `Object.defineProperty()` function may be new to you, it's a JavaScript equivalent of the object-oriented concept of encapsulation.

DEFINITION Encapsulation is the process of hiding a class's internal data by restricting direct access to the data. In object-oriented programming, this is done by creating a private class variable, and a public property that allows you to access the private variable through `get()` and `set()` functions. The private variable is sometimes referred to as a private backing variable.

In defining our custom UI controls, each property needs to be encapsulated. The code in listing D.6 demonstrates how we do this by providing a private variable in the constructor: `this._items`, and by encapsulating it with the `Object.defineProperty()` function. You'll notice that the property definition has both a `get` and `set` function, which provide access to the internal `this._items` property.

TIP You can name the private backing variable anything, but we prefer to name it the same as our externally-facing property name with a preceding underscore. For example, the `items` property has a private backing variable named `_items`. Using a naming convention like this can make your code easier to follow.

That's it. We've created the `items` property, so we can now pass in an observable array to our custom UI control via XML: `<shared:ScrapbookList items="{{ listItems }}" />`.

But, wait. If you've got an eye for detail, you may have noticed we didn't explain the `this.bindData()` function call in the `items` property definition. The `bindData()` function plays an important role in telling our control how to use the data passed into the `items` property. Let's take a closer look in the next step.

STEP 4: CONFIGURE CUSTOM DATA-BINDING FOR THE CONTROL'S UI ELEMENTS

Each time our custom UI control's `items` property is set, we need to tell our control what to do with the data. Our assumption is the `items` property will be set to an observable array of scrapbook items. The items will be displayed in the list view created earlier in the chapter. All that's left is to connect the dots: data bind the list view to the observable array passed into the `items` property. Listing D.7 outlines the `bindData()` function, which performs the list view data binding.

Listing D.7 Binding the list view to the items property

```
function ScrapbookList() {
    super.call(this);

    this._items; #A
    ...
    #B
```

```
this.bindData = function () {
    listView.bind({
        sourceProperty: "$value",
        targetProperty: "items",
        twoWay: "true"
    }, this._items);
};

#A Private backing variable for the items property and observable array of scrapbook items
#B Additional constructor code intentionally removed
```

The `bindData()` function is defined in the custom UI control's constructor and uses the `bind()` function to data bind the observable array held inside of `this._items` to the `items` property of the list view.

NOTE The list view's data binding source property is set to `$value`, which is something you haven't seen yet. Typically, data binding uses a binding source (like an observable or observable array) and links together a property from the binding source to property on a UI element. For example, a label's text field may be data bound to an observable's text property. In our custom control, our binding source is an observable array, and it is the collection of scrapbook items, meaning there is not property on the observable that we can bind the list view's `items` property to. Because the observable is the object we want to bind to, the special `$value` property is used. This tells the list view that it will be binding directly to the value of `this._items`, and not a property on the object.

Great work! You've created a custom UI control based on a grid layout, added child UI elements, a custom property that participates in data binding. Let's move on to the last step, where you'll learn how to create a custom event in response to a user tapping a list view item.

STEP 5: CONFIGURE CUSTOM EVENTS FOR THE CONTROL'S UI ELEMENTS

In many of the previous chapters, you've learned about the various UI element events, such as the page's loaded event and the button's tap event. Custom UI controls are like these UI elements you've learned about because they too can have events. But, the key difference is that you must create the events in custom UI controls.

You might be wondering what is the benefit of creating custom UI control events, especially because custom UI controls are a collection of UI elements (and these elements already have events defined). The answer lies in the details of how a custom UI control works when it is used on a page.

When a custom UI control is used on a page, the detailed implementation of the control is hidden from the page. For example, consider the scrapbook list we're developing into a custom UI control. When we place the custom UI control on a page, the page doesn't directly know about the internal structure of the control. It doesn't know it's built on top of a grid layout, and that the grid layout contains a stack layout, list view, list view item template, image, and label. In fact, all it knows is that an item's property exists and that it should be data bound to an observable array.

Figure D.4 shows the relationship between the list view, custom UI control, and a page further.

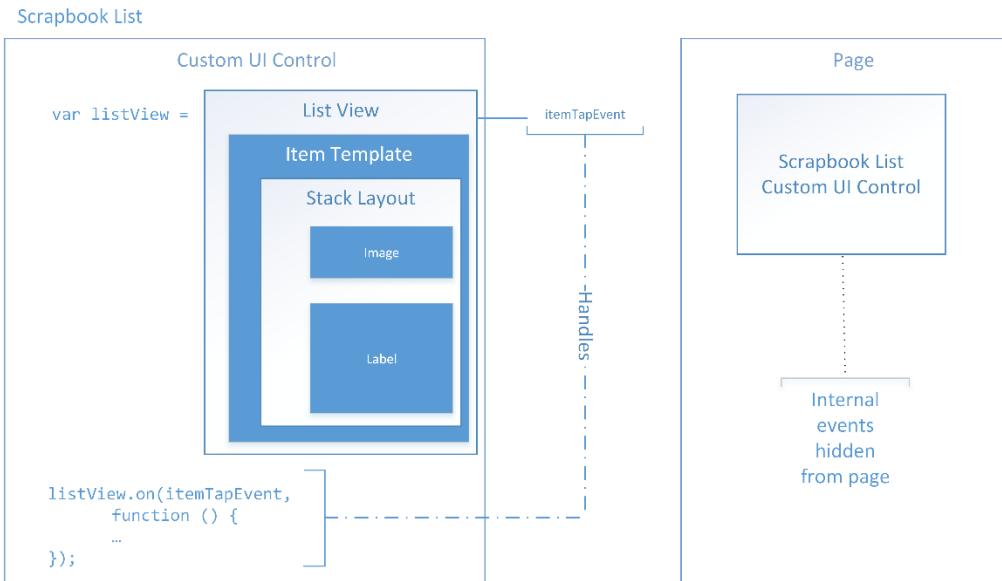


Figure D.4 Events raised by internal elements of custom UI controls are hidden from the pages using the custom UI controls.

On the left, the custom UI control is defined with an internal list view (or in other words, the list view's scope is internal to the custom UI control). On the right, the custom UI control is added to a page, showing that internal UI elements can't be accessed. Furthermore, the list view item tap event's scope is also internal to the custom UI control, meaning it too cannot be accessed from a page.

To bridge the gap between the internal control events and events externally available on pages, we need to *bubble* up the internal event to the page by creating a custom event.

DEFINITION Bubbling events is the process by which an event is raised outside of its scope. For example, if an event inside of a custom UI control is raised so it is accessible outside of the control's scope (an on the scope of a page), it is considered bubbling the event up to the page.

Listing D.8 outlines the approach we'll take to bubbling up the list views item tap event.

Listing D.8 Bubbling the item tap event up in the custom UI control

```

function ScrapbookList() { #A
    super.call(this);
    ...
    #B
    listView.on(ListView.itemTapEvent, function(args) { #C
        onItemTap(this, args.index); #D
    }.bind(listView));
    #D
}

```

```

var onItemTap = function(args, index) {
    this.notify({                                #E
        eventName: "itemTap",
        object: this,
        index: index
    });
}.bind(this);                                #F
}
return ScrapbookList;
#A Internal event handlers are defined in the constructor
#B Previous constructor content left out intentionally
#C Bubbling events start with handling the event to bubble
#D Variable this refers to the list view
#E Events are bubbled by notifying others
#F Variable this refers to the custom UI control

```

Like our previous additions to the custom UI control, we worked from within the constructor. Event bubbling is a three-step process:

1. Attaching an event handler to our list view's item tap event
2. Creating a new item tap event by notifying others that that event has occurred
3. Advertising the event by adding an event property to the custom UI control

STEP 5.1: ATTACHING AN EVENT HANDLER TO OUR LIST VIEW'S ITEM TAP EVENT

In chapter 4, you learned how to attach handlers to UI elements by adding the event handler function name to the UI element's declaration. For example, `<ListView itemTap="onItemTap" />` would add an event handler named `onItemTap` (of course you'd also have to define the `onItemTap` function in JavaScript). But, with our custom UI control, there is no XML declaration, so we must do everything in code. The `listView.on()` function accomplishes the same thing.

TIP To attach event handlers to a UI element with JavaScript, use the `on()` function. Every UI element implements this function.

Listing D.9 shows code added to the `ScrapbookList()` function after the list view's creation and before adding the list view to the stack layout.

Listing D.9 Defining the list view's itemTap event handler via JavaScript

```

function ScrapbookList() {
    _super.call(this);

    this._items;
    this._itemTap;

    this.rows = "*";
    this.columns = "*";

    var listView = new ListView();
    listView.className = "list-group";
    listView.itemTemplate = function() {
        var stackLayout = new StackLayout();
        stackLayout.orientation = "horizontal";
    }
}

```

```

stackLayout.bind({
  targetProperty: "className",
  sourceProperty: "$value",
  expression: "isActive ? 'list-group-item active' : 'list-group-item'"
});

var image = new Image();
image.className = "thumb img-circle";
image.bind({
  targetProperty: "src",
  sourceProperty: "image"
});
stackLayout.addChild(image);

var label = new Label();
label.className = "list-group-item-text";
label.style.width = "100%";
label.textWrap = true;
label.bind({
  targetProperty: "text",
  sourceProperty: "title",
  expression: "(title === null || title === undefined ? 'New' : title + '\\\\\'s') +
+ ' Scrapbook Page'"
});
stackLayout.addChild(label);

return stackLayout;
};

listView.on(ListView.itemTapEvent, function(args) { #A
  onItemTap(this, args.index); #A
}.bind(listView)); #A

this.addChild(listView);

this.bindData = function () {
  var bindingOptions = {
    sourceProperty: "$value",
    targetProperty: "items",
    twoWay: "true"
  };
  listView.bind(bindingOptions, this._items);
};

var onItemTap = function(args, index) { #B
  #B
}.bind(this); #B
}

#A Handles the itemTap event of the list view, calling the onItemTap() function
#B We'll implement this function in the next step

```

NOTE You may have noticed the `.bind(listView)` function appended to the item tap event handler in listing D.9. This isn't a special NativeScript syntax, but a JavaScript-specific syntax that you may not have run into in everyday app development. It's ok if `bind()` seems confusing the first time you see it. Just think of it like this: the `bind()` function tells the function it is attached to what the

value of the variable `this` should represent when executed. So, in our example, `function() { onItemTap(this, args.index); } .bind(listView);`, the value of `this` passed into the `onItemTap` function will be our instance of the list view element.

STEP 5.2: CREATING A NEW ITEM TAP EVENT BY NOTIFYING OTHERS THAT THAT EVENT HAS OCCURRED

The second step is to actually bubble up the item tap event by notifying others, which is done by calling `this.notify()`. When `this.notify()` is called, it raises an event with a name specified in the `eventName` property, bubbling up the object and tapped index with the event. Listing D.10 outlines the changes to the `onItemTap()` function we'll make to bubble up the `itemTap` event.

Listing D.10 Bubbling up the itemTap event with the notify() function

```
var onItemTap = function(args, index) {
    this.notify({
        eventName: "itemTap",
        object: this,
        index: index
    });
}.bind(this);
```

You'll also recognize the use of the `bind()` function again in listing D.10, but in this particular circumstance, the variable `this` refers to the instance of our custom UI control.

STEP 5.3: ADVERTISING THE EVENT BY ADDING AN EVENT PROPERTY TO THE CUSTOM UI CONTROL

The third (and final) step is to add an `event` property to the custom UI control.

DEFINITION Event properties are properties on custom UI controls that tell pages the events a custom UI control has. For example, a custom UI control with an `itemTap` event would have an event property named `itemTapEvent` with a value of "itemTap" assigned to it. The name and value of an event property are important because NativeScript uses a convention to detect event properties. When a custom UI control is loaded, the control's properties are examined. Any properties named *Event and assigned to a string value are assumed to be events. When an event property is added to a custom UI control, it can be used in the XML declaration of a custom UI control and data-bound.

Using the new event property concept you just learned, add a property to the `ScrapbookList` object named `itemTapEvent` and assign it a value of "itemTap". This should be added after the declaration of the `ScrapbookList` object, as shown in listing D.11.

Listing D.11 Adding the itemTapEvent event property to the ScrapbookList object

```
var ScrapbookList = (function (_super) {
    global.__extends(ScrapbookList, _super);
    ...
    #A
}) (GridLayout);
ScrapbookList.itemTapEvent = "itemTap"; #B
```

```
exports.ScrapbookList = ScrapbookList;
#A Code intentionally left out to save space
#B By convention, the itemTapEvent property defines an event named itemTap on the ScrapbookList object
```

Finally! We're finished with our first custom UI control. And, the best part about building this custom UI control is that we can reuse these same five steps to build any custom UI control.

Just in case you weren't able to follow along, here's the complete version of the scrapbook list custom UI control (listing D.12).

Listing D.12 Complete code for the scrapbook list custom UI control

```
var GridLayout = require("ui/layouts/grid-layout").GridLayout;
var ListView = require("ui/list-view").ListView;
var StackLayout = require("ui/layouts/stack-layout").StackLayout;
var Label = require("ui/label").Label;
var Image = require("ui/image").Image;

var ScrapbookList = (function (_super) {
    global.__extends(ScrapbookList, _super);

    Object.defineProperty(ScrapbookList.prototype, "items", {
        get: function() {
            return this._items;
        },
        set: function(value){
            this._items = value;
            this.bindData();
        }
    });
}

function ScrapbookList() {
    _super.call(this);

    this._items;

    this.rows = "*";
    this.columns = "*";

    var listView = new ListView();
    listView.className = "list-group";
    listView.itemTemplate = function() {
        var stackLayout = new StackLayout();
        stackLayout.orientation = "horizontal";
        stackLayout.bind({
            targetProperty: "className",
            sourceProperty: "$value",
            expression: "isActive ? 'list-group-item active' :
                'list-group-item'"
        });

        var image = new Image();
        image.className = "thumb img-circle";
        image.bind({
            targetProperty: "src",
            sourceProperty: "image"
        });
    }
}
```

```
});

stackLayout.addChild(image);

var label = new Label();
label.className = "list-group-item-text";
label.style.width = "100%";
label.textWrap = true;
label.bind({
    targetProperty: "text",
    sourceProperty: "title",
    expression: "(title === null || title === undefined ?
        'New' : title + '\\\\\'s') + ' Scrapbook Page'"
});
stackLayout.addChild(label);

return stackLayout;
};

listView.on(ListView.itemTapEvent, function(args) {
    onItemTap(this, args.index);
}.bind(listView));

this.addChild(listView);

this.bindData = function () {
    var bindingOptions = {
        sourceProperty: "$value",
        targetProperty: "items",
        twoWay: "true"
    };
    listView.bind(bindingOptions, this._items);
};

var onItemTap = function(args, index) {
    this.notify({
        eventName: "itemTap",
        object: this,
        index: index
    });
}.bind(this);
}
return ScrapbookList;
}) (GridLayout);
ScrapbookList.itemTapEvent = "itemTap";

exports.ScrapbookList = ScrapbookList;
```

Now that we've built the control, let's integrate it into the scrapbook app!

D.2 Using custom UI controls

Earlier in this chapter, you learned how to use basic custom UI controls in a page by registering an XML namespace and adding the UI element. Let's put this to use by incorporating the scrapbook list custom UI control into our app. First, we'll replace the existing list view UI element on the scrapbook list page with our custom UI control, then incorporate the same custom UI control into the tablet-specific page.

D.2.1 Replacing existing XML code with a custom UI control

As you'll recall from chapter 11, the scrapbook list page had a list view element embedded directly into the page, as shown in listing D.13.

Listing D.13 The scrapbook-page.xml page before replacing the grid layout and list view with a custom UI control

```
<Page loaded="onLoaded">
  <Page actionBar>
    <ActionBar title="Pet Scrapbook" >
      <ActionItem tap="onAddTap" ios.position="right"
        text="Add" android.position="actionBar"/>
    </ActionBar>
  </Page actionBar>

  <GridLayout rows="*" columns="*"
    <ListView class="list-group" items="{{ pages }}" itemTap="onItemTap">
      <ListView.itemTemplate>
        <StackLayout orientation="horizontal" class="list-group-item">
          <Image class="thumb img-circle" src="{{ image }}"/>
          <Label class="list-group-item-text"
            text="{{ title, (title === null || title === undefined ?
              'New' : title + '\s') + ' Scrapbook Page' }}"/>
        </StackLayout>
      </ListView.itemTemplate>
    </ListView>
  </GridLayout>
</Page>
```

Now that we have the scrapbook list custom UI control, we can replace quite a bit of the XML markup on this page with a single custom UI control element. Listing D.14 shows this transformation, which adds the necessary XML namespace declaration and replaces the grid layout and list view elements.

Listing D.14 The scrapbook-page.xml page after replacing the grid layout and list view with a custom UI control

```
<Page
  xmlns:shared="views/shared/scrapbook-list/scrapbook-list" #A
  loaded="onLoaded">
  <Page actionBar>
    <ActionBar title="Pet Scrapbook" >
      <ActionItem tap="onAddTap" ios.position="right"
        text="Add" android.position="actionBar"/>
    </ActionBar>
  </Page actionBar>

  <shared:ScrapbookList items="{{ pages }}" itemTap="onItemTap" /> #B
</Page>
```

#A XML namespace declaration points to the location of the custom UI control

#B Custom UI control replaced grid layout and list view

In the page element, we added the XML namespace declaration. It does two things: points to the relative location of the scrapbook list custom UI control (`views/shared/scrapbook-list/scrapbook-list`) and creates a namespace name (`shared`) that can be used as a type of alias elsewhere in the page. Once the namespace declaration has been added, we could replace the grid layout and list view with the custom UI

control. You'll notice we were able to use the same data binding object (`pages`) as the previous list view had used, and the same event handler for the `itemTap` event.

With these minimal changes, we're finally ready to reload the Pet Scrapbook. If you've been following along, fire it up, navigate to the scrapbook list page, and look (figure D.5)

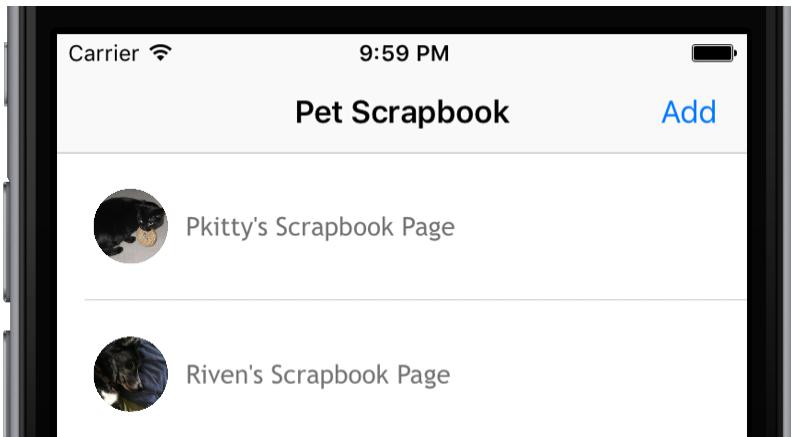


Figure D.5 The scrapbook list page of the Pet Scrapbook, with a custom UI replacing the grid layout and list view elements.

Well, it looks the same. This isn't a let-down: we promise, it's a good sign. We replaced the grid layout and list view perfectly with minimal code changes. Great work!

D.2.2 Adding a custom UI control to a new page

Although we've used our custom UI control once, you may still feel a bit let-down because that was *a lot* of effort. Don't worry, we're about to see our hard work pay us back. Let's get back to the tablet-specific page and add the scrapbook list control.

We're about to add quite a bit of functionality to the tablet-specific page, and before it gets too confusing, we'll step back and get an idea of what we're trying to build. Figure D.6 shows the tablet-specific version of the Pet Scrapbook at the end of chapter 11.

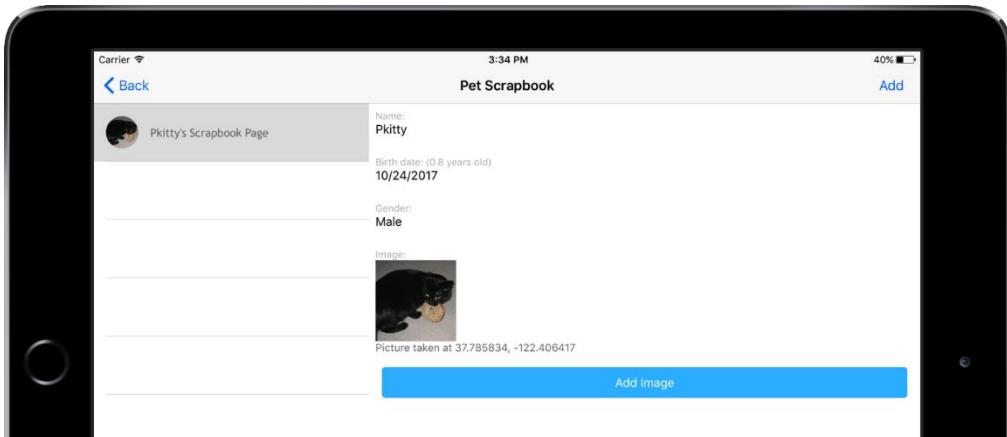


Figure D.6 The Pet Scrapbook at the end of Chapter 11, showing a list view on the left and the details view on the right.

We'll be building the same page, but with reusable custom UI controls instead of copying, pasting, and modifying code like we did in chapter 11. In summary, the tablet version will identical the existing app, and will be composed of three features:

9. A list of scrapbook pages. The left side of the screen will display a list of scrapbook pages, reusing the functionality we've defined already in the *scrapbook-page.xml* file.
10. A detailed view of a single scrapbook page. When a user selects a page from the list, the right side of the screen will display the detailed view of the selected page, reusing the UI and business logic from the *scrapbookUpdate-page.xml* file. From the detailed view the user will be able to save changes made to the scrapbook page.
11. Ability to add a new scrapbook page. Users will be able to add new pages to the list of scrapbook pages and update the page's content from the detailed view on the right.

CREATING THE BASE XML CODE STRUCTURE

Let's start by creating the structure for the new page, leaving a space for the list and details areas. Listing D.15 outlines the base structure of the tablet-specific scrapbook page.

Listing D.15 Initial structure of the *scrapbook-page.land.minWH600.xml* page

```
<Page loaded="onLoaded">
<Page actionBar>
    <ActionBar title="Pet Scrapbook" >
        <ActionItem tap="onAddTap" ios.position="right"
            text="Add" android.position="actionBar"/>
    </ActionBar>
</Page actionBar>
<GridLayout rows="*" columns="*, 2*" >
    #A
    <GridLayout rows="*" columns="*" col="1" >
        <StackLayout> #B
        #B
    </GridLayout>
</GridLayout>
```

```

        </StackLayout> #B
    </GridLayout>
</GridLayout>
</Page>
#A The scrapbook list code goes here
#B The detailed view code goes here

```

This is the same code you saw in chapter 11, displaying the app's name and an action item that adds a new scrapbook page to the list of pages. The remainder of the page is organized with a single grid layout with a single row and two columns. Although you don't see the actual XML code for the list and detailed view in listing D.15, we've called out the location of each. The scrapbook list will be placed in the first column, and consume a third of the screen's width. The detailed view of a selected page will take up the right two-thirds of the screen and be wrapped inside of an additional grid layout and stack layout.

BUILDING OUT THE JAVASCRIPT CODE BASE

To wrap up the base structure of the page, let's lay down the JavaScript code to go along with the XML code we just added by starting with the page's loaded event (listing D.16).

Listing D.16 Handling the loaded event in the scrapbook-page.land.minWH600.js file

```

var observable = require("data/observable");
var observableArray = require("data/observable-array");

exports.onLoaded = function(args) {
    var page = args.object;
    var scrapbook = new observable.fromObject({
        pages: new observableArray.ObservableArray(),
        selectedPage: null
    });

    page.bindingContext = scrapbook;
};

```

Once again, you'll recognize this code from chapter 11. When the tablet-specific page loads, we establish a binding context for the page, assigned to an observable named *scrapbook*. The scrapbook observable will be used to track two things: an observable array of scrapbook pages (the *pages* property) and the selected scrapbook page (the *selectedPage* property).

With the loaded event added, let's turn our attention to the action bar item that adds a new scrapbook page. Now that we have an observable array (*pages*), we'll add a function to handle the tap event and add a scrapbook page to the *pages* observable array (listing D.17).

Listing D.17 Adding a scrapbook page when the action bar action item is tapped

```

exports.onAddTap = function(args) {
    var page = args.object;
    var scrapbook = page.bindingContext;

    scrapbook.pages.push(new scrapbookPageModel(scrapbook.pages.length));
};

function scrapbookPageModel(id) {
    var model = new observable.fromObject({

```

```
        id: id,
        title: null,
        birthDate: null,
        gender: null,
        image: null,
        lat: null,
        long: null,
        isActive: false
    });
}

model.calcAge = function(birthDate) {
    var now = Date.now();
    var diff = Math.abs(now - birthDate) / 1000 / 31536000;

    return diff.toFixed(1);
};

return model;
}
```

We're not going to explain this listing in detail, because you've seen similar code in chapter 11.

That's the last of the base code structure for the tablet-specific page. It feels a little anti-climactic, because there's not a lot to show for our work, but stick with us. Laying down this structure up front will make it easy for us to add a reusable UI component for the list and details pages. We'll be setting this code aside for now but revisiting it once we've added the scrapbook list custom UI control.

REUSING A CUSTOM UI CONTROL

We've already used the scrapbook list custom UI control once, so let's not hesitate in jumping right into the XML code. Listing D.18 updates the tablet-specific page.

Listing D.18 Adding the scrapbook list custom UI control to the tablet-specific page

```
<Page
    xmlns:shared="views/shared/scrapbook-list/scrapbook-list"
    loaded="onLoaded">
    <Page.actionBar>
        <ActionBar title="Pet Scrapbook" >
            <ActionItem tap="onAddTap" ios.position="right"
                text="Add" android.position="actionBar"/>
        </ActionBar>
    </Page.actionBar>
    <GridLayout rows="*" columns="*, 2*" >
        <shared:ScrapbookList items="{{ pages }}" itemTap="onItemTap" />
        <GridLayout rows="*" columns="*" col="1">
            <StackLayout>
                </StackLayout>
            </GridLayout>
        </GridLayout>
    </Page>
```

The additions made to this page are the exact changes we made to the other scrapbook page: adding a namespace declaration and the custom UI control element.

BINDING DATA TO THE TABLET-SPECIFIC SCRAPBOOK LIST

We can't call the left-side of the scrapbook page done until we've displayed the same scrapbook list data, so let's borrow some code from the existing scrapbook list page and add it to the tablet-specific JavaScript file (listing D.19).

Listing D.19 Adding the scrapbook list custom UI control to the tablet-specific page

```
var view = require("ui/core/view"); #A
var fileSystemService = require("~/data/fileSystemService");

exports.onLoaded = function(args) {
    var page = args.object;

    var scrapbook = new observable.fromObject(                      #B
        { pages: new observableArray.ObservableArray() });          #B
    var pages = fileSystemService.fileSystemService.getPages(); #B

    if (pages.length !== 0) {                                       #B
        pages.forEach(function (item) {                            #B
            var model = new scrapbookPageModel();                 #B

            model.id = item.id;                                    #B
            model.title = item.title;                            #B
            model.gender = item.gender;                         #B
            model.birthDate = item.birthDate;                   #B
            model.image = item.image;                           #B
            model.lat = item.lat;                                #B
            model.long = item.long;                             #B

            scrapbook.pages.push(model);                        #B
        });
    }
    else {
        scrapbook = new observable.fromObject({                #B
            pages: new observableArray.ObservableArray()     #B
        });
    }

    page.bindingContext = scrapbook;
};

exports.onItemTap = function(args) {
    var page = args.object;
    var scrapbook = page.bindingContext;

    scrapbook.pages.forEach(function (item) {                  #C
        item.set("isActive", false);                         #C
    });

    scrapbook.set("selectedPage",                          #C
        scrapbook.pages.getItem(args.index));             #C
    scrapbook.selectedPage.set("isActive", true);         #C
}
#A We'll need the view module to get a reference to scrapbook list element by id
#B We borrowed the same event handler and file system loading code from the other page using scrapbook list
control
```

#C isActive is used to style the selected list view item

There's a lot of code here, but truthfully, you've seen most of it before, so we won't dive in too deep.

In the page loaded handler, we used the view module to get a reference to our custom UI control element and added a handler to its item tap event. We then added the same code from the *scrapbook-page.js* page loaded handler to get our list of scrapbook data from the file system and bind it to the page.

We also copied the `onItemTap()` function from the scrapbook list page, but made some slight modifications. On the other page, this function navigated the user to the details page, but the details page will be displayed on the right side of this page. Instead of navigating, we set the selected page and set the selected list view item with the `isActive` flag.

NOTE You may have forgotten what this flag does, so here's a quick reminder. Each list view item's

template contains a stack layout with the CSS class property bound to the `isActive` flag:

```
<StackLayout class="{{ isActive ? 'list-group-item active' : 'list-group-item' }}" />
```

When true, the list item is styled with the `active` class, making it stand out.

Let's load up your app in an emulator or simulator and check it out. Don't forget to select a tablet as the device to see your hard work in action (figure D.7).

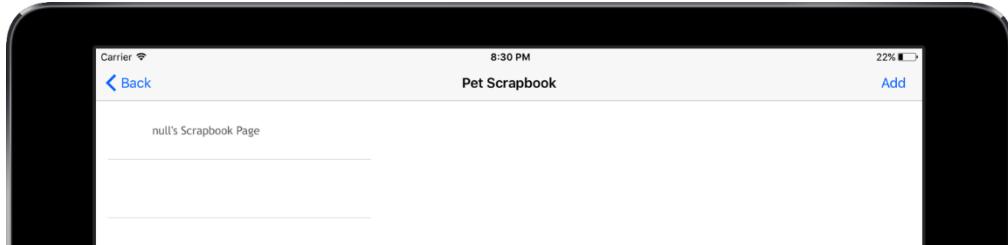


Figure D.7 The custom UI scrapbook list control added to the tablet-specific page

Great work! We've created a custom UI control to replace the scrapbook list. Now, if we ever want to change the styling or functionality of the scrapbook list, we can make the changes in a single place (within the custom UI control), and both pages will benefit. For the complete code shown in the chapter, visit <https://github.com/mikebranstein/TheNativeScriptBook/tree/master/AppendixD>.

D.3 Summary

In this appendix, you learned the following:

- How to create simple XML-based custom UI controls.
- How XML namespaces work and how the `xmlns` property is used to add a custom UI control to a page.
- How to inherit from a layout UI element to create a JavaScript-based custom UI control.
- How the `Object.defineProperty()` function is used to create a public property that can be used for data binding.

- How to bubble up custom events from a UI element within a custom UI control using the `notify()` function.

D.4 Challenge

You may have noticed that this appendix didn't build out the scrapbook details custom UI control that can be used on both the `scrapbookUpdate-page.xml` and tablet-specific pages. Building custom UI controls can be a bit tricky, but extremely rewarding once completed. You have learned everything you need to know to build a custom UI control for the details page, so have at it! If you get stuck, or just want to see it in action, feel free to check out the complete code at <https://github.com/mikebranstein/TheNativeScriptBook/tree/master/AppendixD>.

Have fun!