

# IONIC

**SUCCINCTLY**

*BY* **ED FREITAS**

# Ionic Succinctly

---

By  
Ed Freitas

Foreword by Daniel Jebaraj



Copyright © 2018 by Syncfusion, Inc.

2501 Aerial Center Parkway

Suite 200

Morrisville, NC 27560

USA

All rights reserved.

**Important licensing information. Please read.**

This book is available for free download from [www.syncfusion.com](http://www.syncfusion.com) on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from [www.syncfusion.com](http://www.syncfusion.com).

This book is licensed for reading only if obtained from [www.syncfusion.com](http://www.syncfusion.com).

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

**Technical Reviewer:** James McCaffrey

**Copy Editor:** Courtney Wright

**Acquisitions Coordinator:** Tres Watkins, content development manager, Syncfusion, Inc.

**Proofreader:** Jacqueline Bieringer, content producer, Syncfusion, Inc.

# Table of Contents

<b>The Story Behind the Succinctly Series of Books.....</b>	<b>7</b>
<b>About the Author .....</b>	<b>9</b>
<b>Acknowledgements .....</b>	<b>10</b>
<b>Introduction.....</b>	<b>11</b>
<b>Chapter 1 Setting up Ionic.....</b>	<b>13</b>
Who is this book for? .....	13
Why a complete framework rewrite?.....	13
Nice-to-haves .....	14
Installing Ionic-Angular .....	14
Creating a new Ionic project .....	16
Testing the scaffolded app .....	18
Live or Hot Reload.....	20
Ionic View and Ionic Pro .....	20
Deploying to Ionic Pro .....	24
Summary.....	29
<b>Chapter 2 Project Structure.....</b>	<b>30</b>
Quick intro .....	30
The \src folder .....	30
The \src\app folder .....	31
The \src\assets folder .....	32
The \src\pages folder.....	32
The \src\theme folder.....	33
The files within the \src folder .....	33
The \node_modules folder .....	35

The \www folder .....	35
The \ (root) folder.....	35
Summary.....	35
<b>Chapter 3 Starting the App.....</b>	<b>36</b>
Quick intro .....	36
Overview .....	36
Navigation stack.....	36
Main difference from Ionic for AngularJS .....	37
App navigation overview.....	37
Creating the app's main page.....	38
Visual Studio Code Ionic plugins .....	41
Creating pages with the CLI .....	41
Wiring up the new pages .....	43
Summary.....	46
<b>Chapter 4 Building the App .....</b>	<b>47</b>
Quick intro .....	47
Before we start .....	47
The Library page .....	47
Making a nicer card.....	50
Updating the Library page .....	52
Updating the Paths page .....	56
Updating the Books page .....	60
The Book Overview page .....	63
NavParams .....	63
Updating the Book Overview page .....	65
Adding navParams to the Books page.....	67

Adding navParams to the Paths page .....	72
Updating the Book Detail page .....	76
Full source code .....	78
Summary .....	78
<b>Chapter 5 Further Resources .....</b>	<b>79</b>
Quick intro .....	79
Exploring the documentation .....	79
Native plugins .....	80
Search bar .....	81
Adding the search functionality .....	83
Full, updated source code .....	90
Other topics to explore .....	90
Summary .....	91

# The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President  
Syncfusion, Inc.

**S**taying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

## Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

## The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

## The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

## Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

## Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

## Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at [succinctly-series@syncfusion.com](mailto:succinctly-series@syncfusion.com).

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!





# About the Author

Ed Freitas is a consultant on software development applied to customer success, mostly related to financial process automation, accounts payable processing, and data extraction.

He's enjoys playing soccer, running, traveling, life hacking, learning, and spending time with his family. You can find him at <http://edfreitas.me>.

# Acknowledgements

Many thanks to all the people who contributed to this book and the amazing [Syncfusion](#) team that helped it become a reality—especially Jacqueline Bieringer, Tres Watkins, Darren West, and Graham High.

The manuscript manager and technical editor thoroughly reviewed the book's organization, code quality, and overall accuracy—Jacqueline Bieringer from Syncfusion and [James McCaffrey](#) from [Microsoft Research](#). Thank you all.

This book is dedicated to Lala, whose love and support is always there for me, and my little princess angel *Mi Chelin*, who inspires me every day and lights my path ahead—God bless you always.

# Introduction

The [Ionic Framework](#) is a free and open-source SDK for developing amazing native, cross-platform mobile apps (for both Android and iOS) and Progressive Web Apps with ease, using familiar web technologies like HTML, JavaScript, and CSS.

Historically, when mobile app developers wanted to deploy their application on multiple platforms, they had to write one version for Android using Java, another for iOS using Objective-C, and maybe even a specific version for Windows Mobile using C#.

This multiversed approach is not well suited for code reusability or maintenance, and to some extent is rather inefficient in terms of resource management, with few companies being able to afford such an endeavor.

Ionic, on the other hand, enables developers to build an app once—using web development tools that most are already familiar with—to deploy to multiple platforms.

Ionic is not the only framework in the market that allows developers to write code once and deploy to multiple platforms. Others, such as [Xamarin](#) and [NativeScript](#), can also achieve this; however, one of Ionic's key differentiators is that it has always been built (since version 1) on top of [Angular](#). Ionic 1 was built on top of [Angular 1](#), Ionic-Angular on top of Angular 2, and so on.

Not only does Ionic allow developers to reuse their knowledge of general web technologies, but it also gives developers a chance to specifically leverage their existing know-how of the Angular framework—which is beneficial for enjoying all the usability and performance improvements of Angular 2+.

Also, by supporting Angular, Ionic developers are not only part of the thriving Ionic community, but also part of the broader Angular one as well—this is awesome.

Another key facet of Ionic is that it is capable of seamlessly integrating with [Cordova](#), which enables developers to access and use native features on the device through JavaScript or [TypeScript](#) code.

In a nutshell, Cordova—which is specifically designed to be used for mobile apps being built using HTML, JavaScript, and CSS—is a JavaScript API that wraps itself around native APIs, allowing developers to use features such as geolocation, the device camera, the accelerometer, vibrating the device, and much more. All these features have native APIs that must be invoked, so Cordova's job is to standardize these calls and provide a set of JavaScript classes and methods that seamlessly wrap around these native API functionalities, so they can be invoked from JavaScript code.

Apps built with Cordova are truly native mobile apps that can be deployed to an app store—not mobile-optimized web apps that look good on mobile web browsers. Further, Cordova has been around for years, it has a huge and thriving ecosystem, it is stable and mature, and it is widely used—which is one of the reasons it is used inside the Ionic framework, through the Ionic Native library.

Ionic-Angular, which is the specific version of the framework we'll cover throughout this book, is a complete rewrite of the very successful Ionic 1 framework.

To summarize, Ionic is an open source SDK/framework that rides over the AngularJS web application framework and the Apache Cordova cross-platform mobile app framework, and executes in a Node.js runtime environment. Ionic is programmed using mostly TypeScript language and the JSON data format.

The goal of this book is to give you a good start with the Ionic-Angular framework, so that you can jump straight into mobile app development with ease, using your existing web development experience. Some basic knowledge of Angular will help you make the most out of this book.

This book will be quite hands-on, and the examples should be easy and fun to follow. We'll be using [Visual Studio Code](#) as our editor of choice, and TypeScript as our programming language.

So, without further ado, let's dive right into the fascinating Ionic-Angular ecosystem. Thanks for reading!

# Chapter 1 Setting up Ionic

## Who is this book for?

This book is aimed at developers who have no previous experience with Ionic—even no previous experience with Ionic 1. However, it does help a bit if you do have some previous experience with the framework, Angular, or TypeScript—although these are not must-haves.

This book is not a migration manual from Ionic 1, and it's perfectly alright if you are starting out with Ionic-Angular (version 2 or higher of the Ionic framework) from scratch.

## Why a complete framework rewrite?

There are several reasons why the Ionic team did a complete rewrite of the Ionic framework. Probably the most important one is that Ionic-Angular is built with the most recent versions of Angular, which is a massive departure from Angular 1 (also known as AngularJS).

Basically, building on top of the latest Angular versions enabled Ionic-Angular to benefit from various improvements, including increases in performance and a more standardized way of writing and building apps, especially when using TypeScript—like being able to use dependency injection, components, and modules.

For instance, version 1 of the Ionic framework used UI router for navigation. Even though UI router is quite powerful, in Ionic-Angular this has been taken a step further in order to achieve a more natural way of moving from screen to screen.

Another awesome feature in Ionic-Angular is that the framework is capable of detecting if your app is running on an Android or iOS device, and thus render the respective platform theme automatically.

Furthermore, the Ionic team continues to improve the way the internal build system works by using tools like [Gulp](#) for building, serving, and performing live reload when a file is changed.

Also, by using TypeScript for compilation, [SASS](#) for CSS extension and optimization, and NPM for package management, the Ionic team has made this newest version a truly modern and enjoyable mobile app development platform. They have looked at optimizing as much as they could without compromising efficiency and the overall developer experience.

Some of these optimizations are related to better caching, faster navigation between screens, and also the use of internal virtual lists to enable scrolling of very long lists—in order to have native-like performance.

All these improvements make Ionic-Angular a wonderful new version of the framework—one that many developers love.

Ionic-Angular is a framework that is in constant evolution—at an incredibly fast pace. So it is highly likely that between the time this book was written and the time you read it, some code or project structure for a new scaffolded app might be slightly or quite different from what is described here. Nevertheless, you should be able to easily find your way around, as it's been written and reviewed to be as up-to-date as possible.

## Nice-to-haves

In order to make the most of this book, there are a couple of technologies that would be good for you to be familiar with: TypeScript and a recent version of Angular. All the code in this book will make use of them. TypeScript will be installed when installing Ionic, which is really handy.

But don't worry—not having any previous knowledge of them is not a showstopper. If you are a software developer experienced with other web technologies, you should also be able to pick this up fairly quickly, and without too much effort.

Luckily, the *Succinctly* series has you covered, and there's an awesome book written by Steve Fenton called [TypeScript Succinctly](#) that can get you up-to-speed with this technology in no time.

Furthermore, Angular 1 is covered extensively by [AngularJS Succinctly](#), which is brilliantly authored by Frederik Dietz, and Angular 2 is nicely and easily explained in [Angular 2 Succinctly](#) by Joseph D. Booth. Some familiarity with Angular's dependency injection and how data binding works is beneficial.

If you feel that you need to refresh your TypeScript and Angular know-how—or get started from scratch—these are magnificent resources that I highly encourage you to have a look at.

## Installing Ionic-Angular

In order to really get started, the next thing we need to do is install Ionic-Angular. All developments done in Ionic are based around a [Node.js](#) command line tool called Ionic CLI, so first, we need to download Node.js and grab the appropriate installer for our operating system. Whether you're on a Windows PC or Mac, Node.js will work just fine.

Node.js® is a JavaScript runtime built on [Chrome's V8 JavaScript engine](#). Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, [npm](#), is the largest ecosystem of open source libraries in the world.

Important security releases, please update now!

## Download for Windows (x64)

8.9.4 LTS

Recommended For Most Users

9.3.0 Current

Latest Features

[Other Downloads](#) | [Changelog](#) | [API Docs](#)   [Other Downloads](#) | [Changelog](#) | [API Docs](#)

Or have a look at the [LTS schedule](#).

Figure 1-a: The Node.js Home Page

On the Node.js home page, download the appropriate installer for your operating system—in my case, I've downloaded the LTS (Long Term Support) version recommended for most users.

The Node.js website has detected that I'm on a Windows 64-bit machine, and has presented me with the option to download the appropriate runtime for Windows 64 bits.

Installing Node.js basically consists of downloading the installer package and clicking through the prompts in order to get it installed quickly—it's as simple as that.

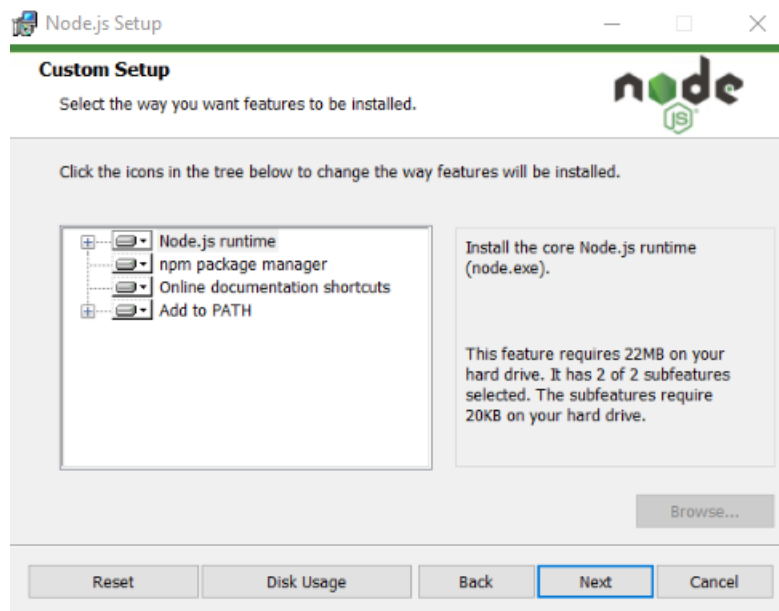
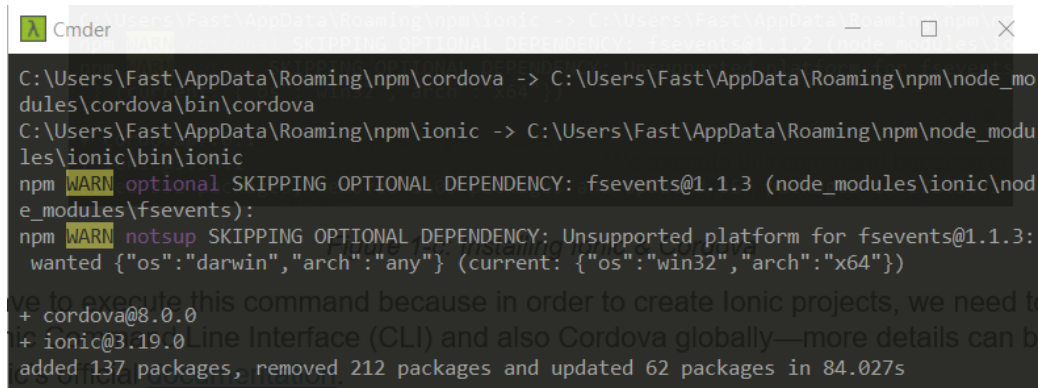


Figure 1-b: The Node.js Installer on Windows

With Node.js installed, we can focus on getting Ionic-Angular up and running. In order to do that, open the command prompt as an *Admin* user—in my case, I use this lovely [emulator](#) on top of the Windows command prompt. Type in the following command.

Code Listing 1-a: The Ionic and Cordova Installation Command

```
npm install -g ionic cordova
```



```
C:\Users\Fast\AppData\Roaming\npm\cordova -> C:\Users\Fast\AppData\Roaming\npm\node_modules\cordova\bin\cordova
C:\Users\Fast\AppData\Roaming\npm\ionic -> C:\Users\Fast\AppData\Roaming\npm\node_modules\ionic\bin\ionic
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.1.3 (node_modules\ionic\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.1.3: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})
+ cordova@8.0.0
+ ionic@3.19.0
added 137 packages, removed 212 packages and updated 62 packages in 84.027s
```

Figure 1-c: Installing Ionic and Cordova

We have to execute this NPM (Node Package Manager) command, because in order to create Ionic projects, we need to install the Ionic Command Line Interface (CLI), and also Cordova globally. You can find more details on Ionic's official [documentation](#).

Once the command has successfully executed, we are ready to start creating apps with Ionic.

## Creating a new Ionic project

In order to start working with Ionic, we need to create a new Ionic project—this can be done by using the Ionic CLI.

Let's type in the following command in order to start a new project, which we'll build throughout this book. Make sure you run this command from the folder where your project's source code will be stored.

Code Listing 1-b: Creating a new Ionic App

```
ionic start IonicAppSuccinctly sidemenu --ts
```

This command will create a subfolder called **IonicAppSuccinctly**, which will contain all the app's code.

The **--ts** option indicates that we will be using TypeScript as our language of choice instead of the default, JavaScript. The **sidemenu** option indicates that the app will have a swipeable menu layout on the side.

Before the command execution actually starts, you might be prompted to answer if you would like this app with Cordova to target native iOS and Android.



Feel free to respond with **Yes** or **No**—in my case, I’ve chosen **No**. Once you do that, installing the project dependencies might take several minutes.

After the project dependencies have been installed, you might also be asked to install the free Ionic Pro SDK and connect your app. Feel free to respond with **Yes** or **No**—in my case, I’ve chosen **Yes**.

After Ionic Pro has been installed—if you’ve chosen **Yes**—you might be asked to confirm the name of the app that you’d like the CLI to create. In my case, I’ve chosen **ionicAppSuccinctly**.

There are other available project templates besides **sidemenu**—more information can be found in the official Ionic [documentation](#).

When this command first executes, it creates the folder **ionicAppSuccinctly**, then downloads the template information, and finally, goes into the folder and executes **npm install** in order to install all the dependencies required by the project, which are specified in the **package.json** file.

If for some reason the execution of the **npm install** command fails, simply go into the **ionicAppSuccinctly** subfolder and manually execute **npm install** in order to try again.

```
C:\Users\Fast\Documents\Visual_Studio_2015\Ionic_Succinctly\Code\ionicAppSuccinctly
λ npm install
```

Figure 1-d: Running the “npm install” Command Manually

If you take a look inside the **ionicAppSuccinctly** subfolder, you should see the following files (or similar).

Name	Type	Size
.git	File folder	
.sourcemaps	File folder	
node_modules	File folder	
src	File folder	
www	File folder	
.editorconfig	Editor Config Source File	1 KB
.gitignore	Text Document	1 KB
ionic.config.json	JSON File	1 KB
package.json	JSON File	2 KB
package-lock.json	JSON File	173 KB
tsconfig.json	JSON File	1 KB
tslint.json	JSON File	1 KB

Figure 1-e: The **ionicAppSuccinctly** Project Files

At this stage, Ionic has created a project structure for us, which we will use as the foundation to build our app.

Let’s quickly look at the scaffolded folder and file structure by opening the project in [Visual Studio Code](#) (VS Code). If you don’t have it installed, now is probably a good time to do so. You don’t have to necessarily use VS Code, and can use any editor of your choice, such as [Atom](#) or [Sublime](#).

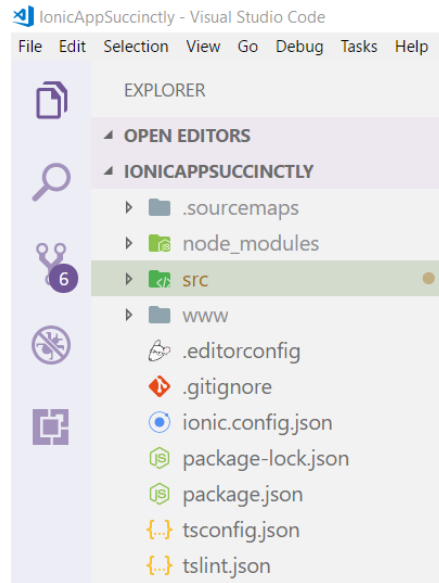


Figure 1-f: The IonicAppSuccinctly Project Structure in VS Code

We'll later explore this project structure in more detail, so for now let's not worry too much about it.

## Testing the scaffolded app

Now that we've created the Ionic project that will eventually become our application, let's see how we can test it using a browser and see how it looks. This is the simplest and most common way of quickly checking things.

We can do this by typing in the following command from the **IonicAppSuccinctly** project subfolder—you'll need to **cd** into **IonicAppSuccinctly**.

*Code Listing 1-c: The "ionic serve" Command*

```
ionic serve
```

This command will basically compile the TypeScript and SCSS (Sassy CSS) code—the one that the Ionic CLI has scaffolded for us—and then launch a web server locally. Let's go ahead and do that to see what happens.

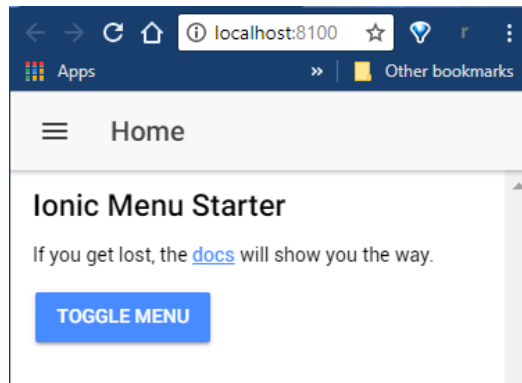


Figure 1-g: The Ionic App in Chrome

We can see that Ionic has started up a web server on **http://localhost:8100** on my machine (the port number might be different on yours).

If your browser's window is fully maximized, you may have noticed that the app looks a bit strange because it's not really designed for a desktop browser.

What you might want to do in that case is press the **F12** key (if you are using Chrome or Edge as your browser)—this will open up the **Developer Tools**. Then, toggle the device toolbar and select a mobile display—this is illustrated in the figure that follows.

You may also choose to dock the Developer Tools panel to either the left or right side of your screen (according to your preference) so you can get more of a feeling that you are actually viewing it on a mobile device.

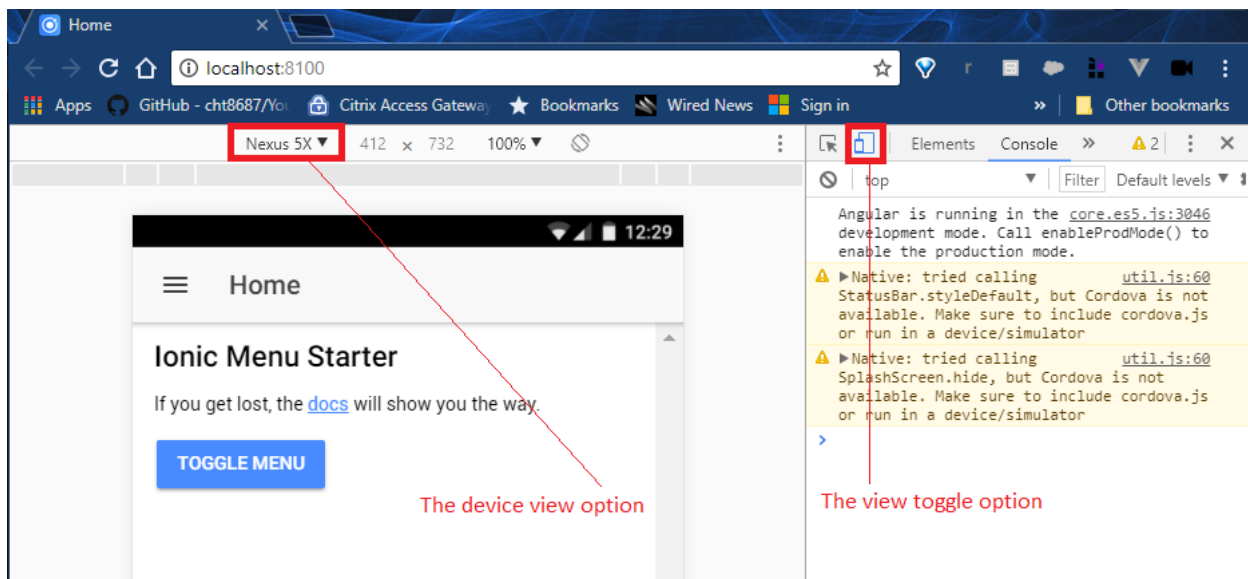


Figure 1-h: Adjusting the View Using Chrome Developer Tools

If you are using other browsers, similar options likely exist—so please check their respective documentation or Help.

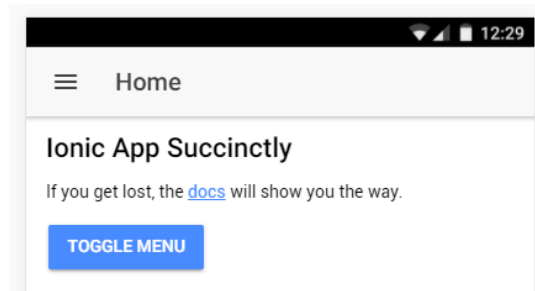
As you can see, the application renders nicely on a Nexus 5X virtual device layout. You can also test it on any other emulated device.

## Live or Hot Reload

One really cool feature that helps a lot during development is what we usually refer to as Live or Hot Reload, which is the ability to reload automatically in the browser a page that has been just edited and saved.

To demonstrate this, I will open up VS Code and go to the `src\pages\home` folder, then open up the `home.html` file and change the default text from **ionic Menu Starter** to **ionic App Succinctly**.

Once you have done this, go ahead and check how the app looks in the browser. Ionic should have automatically reloaded the page, and it should now reflect the changes you just made, after saving the file. Let's have a look.



*Figure 1-i: The App after a Live Reload*

This is a great feature that gives us the ability to quickly test the look and feel of the app and iterate through UI changes quickly. If we want to close the app and web server, we can press **Ctrl + C** from the command line.

As an additional note, throughout most of this book we'll be using the `ionic serve` command in order to test our app, since it is quick and easy to use.

Nevertheless, let's quickly explore how we can test the app with Ionic View.

## Ionic View and Ionic Pro

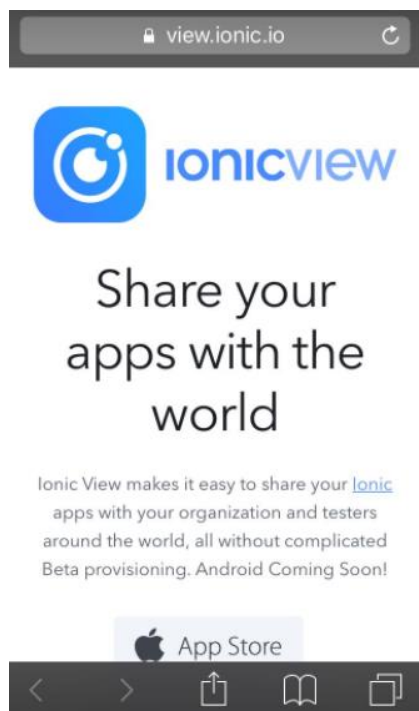
[Ionic View](#) is a tool that allows us to quickly test and run Ionic apps on an actual device without having to deploy the app. It's ideal when you want to share your Ionic app within your organization or with testers without any provisioning.

Ionic View and Ionic Pro are rapidly evolving tools, so it is possible that by the time you read this, some of the screens might look slightly different. In any case, you should be able to find your way around easily with the instructions that follow.

Ionic View is basically a native app that can be installed from the App Store or Google Play, and runs your Ionic app's code on your device.

In order to use Ionic View, we have to install the Ionic View app itself on the device and deploy our Ionic app to Ionic Pro, which you can think of as Ionic's cloud.

From your mobile device, go to the Ionic View [website](#) and install the respective version for either your iOS or Android device. I'll be testing using an iPhone 5.



*Figure 1-j: The Ionic View Site Seen from My iPhone*

I'll install the [iOS version](#) on my device, which has a slightly different look and feel to it than the older (legacy) version, so the screen shots you will see will reflect this UI. The Android version can be found [here](#).

The [older](#) iOS version and the legacy Android [version](#) were designed to work with an older version of Ionic Cloud, compatible with an older version of the Ionic CLI that uses the obsolete **ionic upload** command. This was part of the [old](#) Ionic Cloud product, replaced by Ionic Pro.

I strongly suggest you do not use any legacy versions and follow along with these steps with the latest version of Ionic View—compatible with Ionic Pro.

Once you have the Ionic View app installed on your device, the next thing to do is create an account on [Ionic Pro](#) so we can upload our app to it.

After creating an account and logging in for the first time, you will see a screen similar to the one in Figure 1-k. Select the type of plan to use—I'll be using the **Kickstarter Edition** for the purpose of the app we'll be building throughout this book.

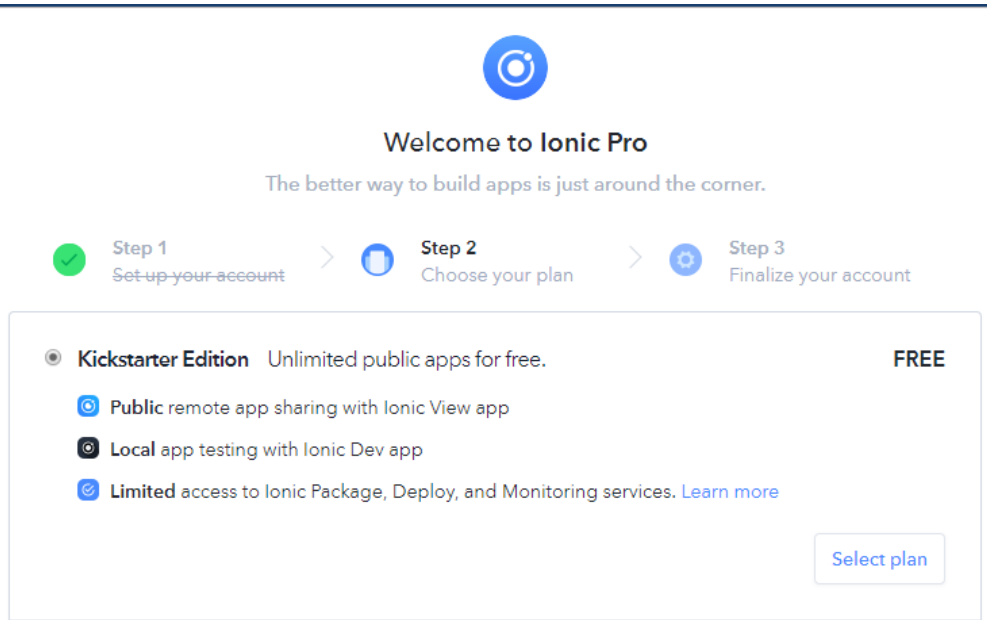


Figure 1-k: Setting Up an Account on Ionic Pro

Once you've selected the type of account, you'll be prompted to create a new app. Click **Create app** in order to finalize the process—with that done, you'll see the following.

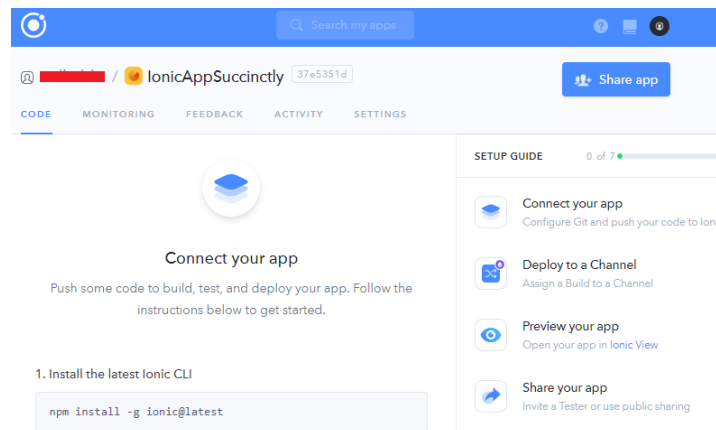


Figure 1-l: The App on Ionic Pro

With our app created in Ionic Pro, all we need to do is link to it and upload the code by following the instructions listed in the previous figure—which in my case, are the following ones.

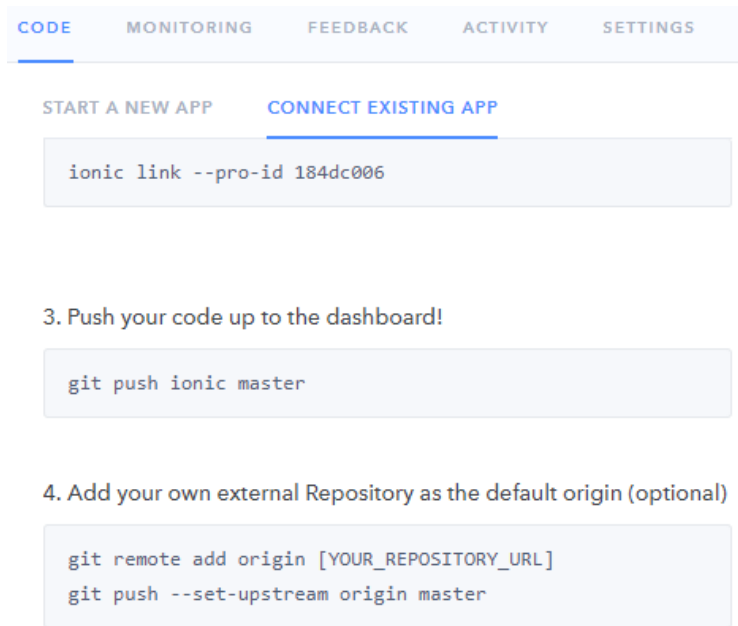


Figure 1-m: Instructions for Connecting an Existing App to Ionic Pro

When executing these commands, run the command line from the same root folder where the `package.json` file is located.

When we execute `ionic link --pro-id...`, our local app is linked to the app repository we've just created on Ionic Pro. This command requires that we enter our Ionic Pro credentials, which are the same ones used when signing up.

Code Listing 1-d: The “ionic link” Command

```
ionic link --pro-id 184dc006
```

You'll likely be asked to configure SSH settings when executing this command. To make the process easy and straightforward, I strongly suggest you select the option **Automatically setup a new SSH key pair for Ionic Pro** when prompted.

An SSH key is required in order to push your app's code securely to Ionic Pro. You can find more info about this topic in Ionic's official [documentation](#).

The following output is what I got when I executed this command on my machine—you might have a similar output.

```
? How would you like to connect to Ionic Pro? Automatically setup new a SSH key pair for Ionic Pro
[INFO] The automatic SSH setup will do the following:
1) Generate a new SSH key pair with OpenSSH (will not overwrite any existing keys).
2) Upload the generated SSH public key to our server, registering it on your account.
3) Modify your SSH config (~/.ssh/config) to use the generated SSH private key for our server(s).

? May we proceed? Yes
> ionic ssh generate C:\Users\Fast\.ssh\ionic\10886_rsa
[INFO] Created ~\.ssh\ionic directory for you.

[INFO] You will be prompted to provide a passphrase, which is used to protect your private key should you lose it. (If
someone has your private key, they can impersonate you!) Passphrases are recommended, but not required.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
[OK] A new pair of SSH keys has been generated!
Private Key (~\.ssh\ionic\10886_rsa): Keep this safe!
Public Key (~\.ssh\ionic\10886_rsa.pub): Give this to all your friends!
[INFO] Next steps:
- Add your public key to Ionic: ionic ssh add ~\.ssh\ionic\10886_rsa.pub
- Use your private key for secure communication with Ionic: ionic ssh use ~\.ssh\ionic\10886_rsa
> ionic ssh add C:\Users\Fast\.ssh\ionic\10886_rsa.pub --use
[OK] Your public key (SHA256:nL+q7HFHplr+Mh7dFit6LZ7NCFpP+omEwltuWlcTf2E) has been added to Ionic!
> ionic ssh use C:\Users\Fast\.ssh\ionic\10886_rsa
--- C:\Users\Fast\.ssh\config with in the number of diff
+++ C:\Users\Fast\.ssh\config
@@ -1,0 +1,3 @@
\ No newline at end of file
+Host git.ionicjs.com
+IdentityFile C:\Users\Fast\.ssh\ionic\10886_rsa
```

Figure 1-n: Connecting our Existing App to Ionic Pro

## Deploying to Ionic Pro

The `ionic link --pro-id` command has been executed successfully.

According to the instructions shown in Figure 1-m, the next thing we should do is execute the `git push ionic master` command.

There's a small but important step missing—which it's assumed that most developers with git experience would automatically notice—before actually executing `git push` (people without any git experience might not notice this at all).

This missing step is that we actually need to commit our code first before pushing it. In order to do this, we need to make a small change to our code. In my case, I'll make a small modification to the `index.html` file found under the `\IonicAppSuccinctly/src` folder by changing the text found on the `<title>` tag.

Once you're done, save the changes. Let's commit the code—we can do this by running the following command: `git add .` followed by `git commit -m "init"`.

Code Listing 1-e: The "git add" Command

```
git add .
```

Code Listing 1-f: The "git commit" Command

```
git commit -m "init"
```



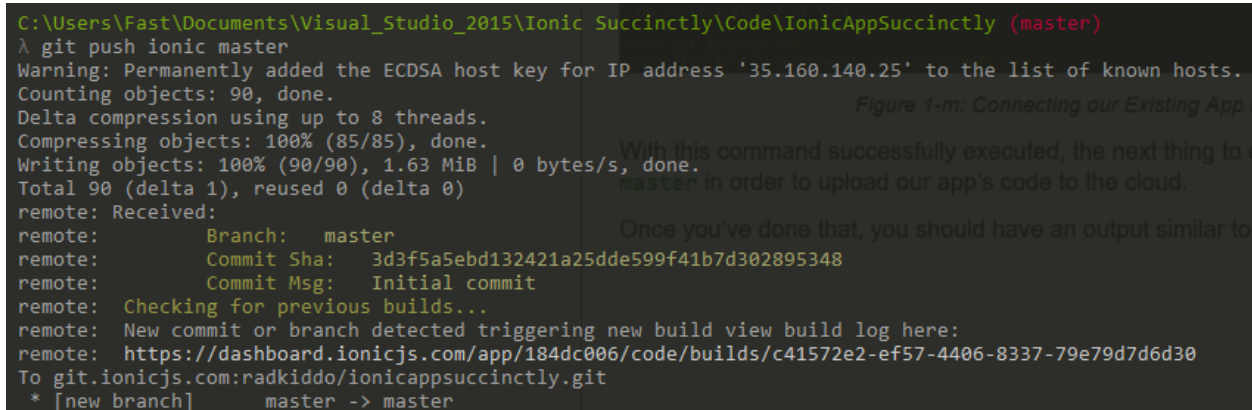
This will commit any changes done to the master git repository—now we are ready to **git push** our code.

So, execute **git push ionic master** in order to upload our app's code to the cloud.

*Code Listing 1-g: The “git push” Command*

```
git push ionic master
```

Once you've done that, you should see an output similar to the following.



```
C:\Users\Fast\Documents\Visual_Studio_2015\Ionic Succinctly\Code\IonicAppSuccinctly (master)
λ git push ionic master
Warning: Permanently added the ECDSA host key for IP address '35.160.140.25' to the list of known hosts.
Counting objects: 90, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (85/85), done.
Writing objects: 100% (90/90), 1.63 MiB | 0 bytes/s, done.
Total 90 (delta 1), reused 0 (delta 0)
remote: Received:
remote:      Branch: master
remote:      Commit Sha: 3d3f5a5ebd132421a25dde599f41b7d302895348
remote:      Commit Msg: Initial commit
remote:      Checking for previous builds...
remote:      New commit or branch detected triggering new build view build log here:
remote:      https://dashboard.ionicjs.com/app/184dc006/code/builds/c41572e2-ef57-4406-8337-79e79d7d6d30
To git.ionicjs.com:radkiddo/ionicappsuccinctly.git
* [new branch]      master -> master
```

*Figure 1-o: Pushing our Existing App Code to Ionic Pro*

Awesome—our app is now being deployed to the cloud and hosted on Ionic Pro.

If we browse to the URL mentioned on the output of the command prompt (as seen in Figure 1-o), we can actually follow the build process taking place—here's mine.

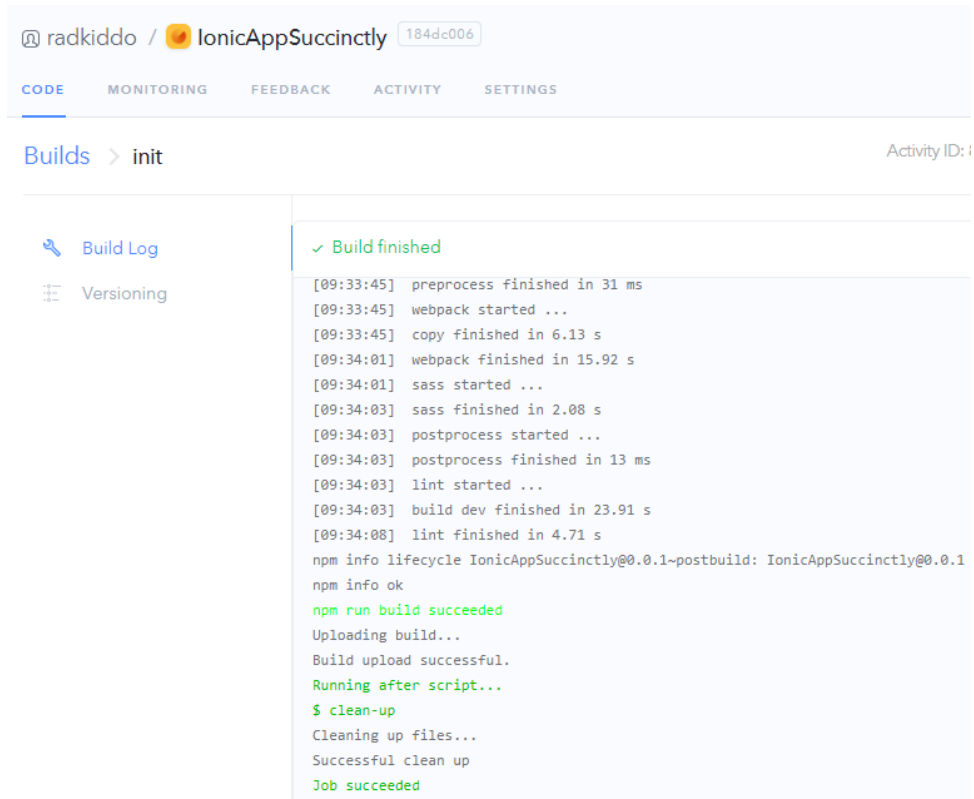


Figure 1-p: The App Code Build Process

We are almost ready to check our app with Ionic View, but before we can do that we need to get the app's shareable ID. You can find it by clicking **Share App** on the Ionic Pro dashboard, and then clicking the **PUBLIC VIEW APP** tab, as seen in the following figure.

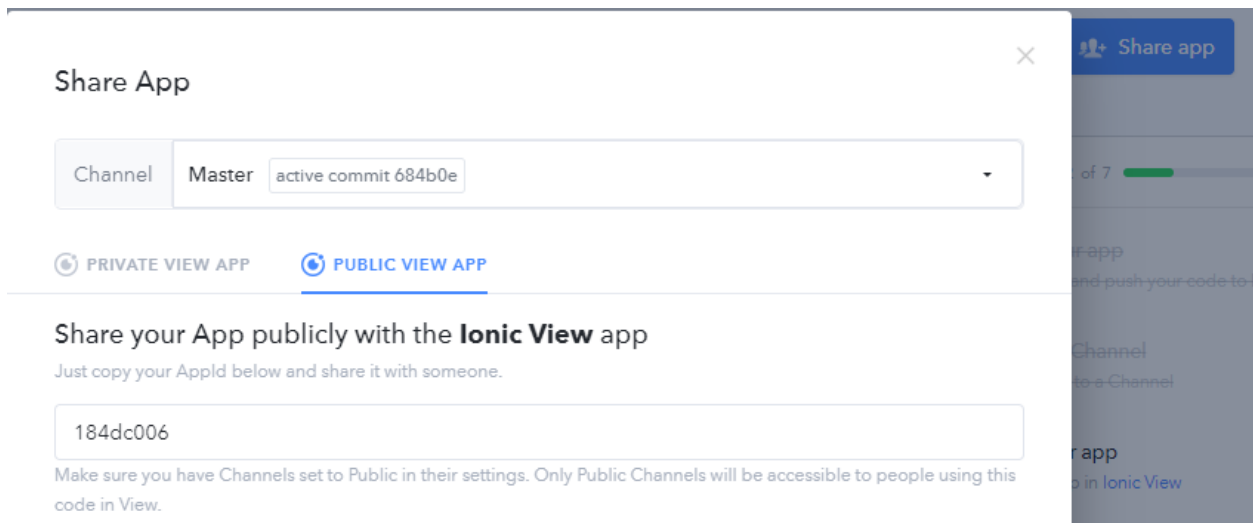


Figure 1-q: The Share App Option

Finally, we also need to make the **Master** channel public. We can do this from the Ionic Pro dashboard by going into **Channels** and clicking on **Master**.

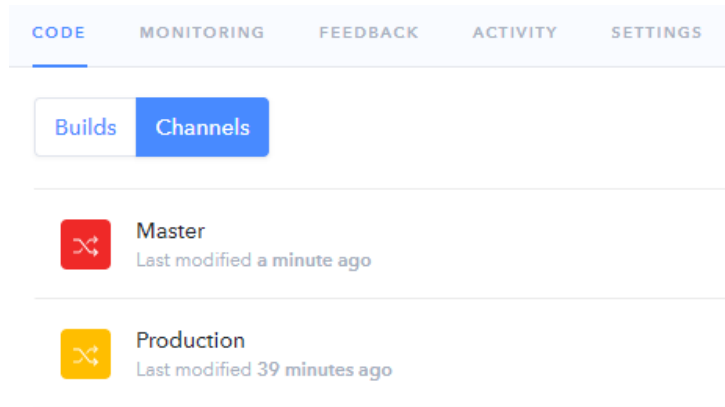


Figure 1-r: Deployment Channels in Ionic Pro

Once you've clicked **Master**, a pop-up window will appear. Go to the **SETTINGS** tab, and then click **Make public**.

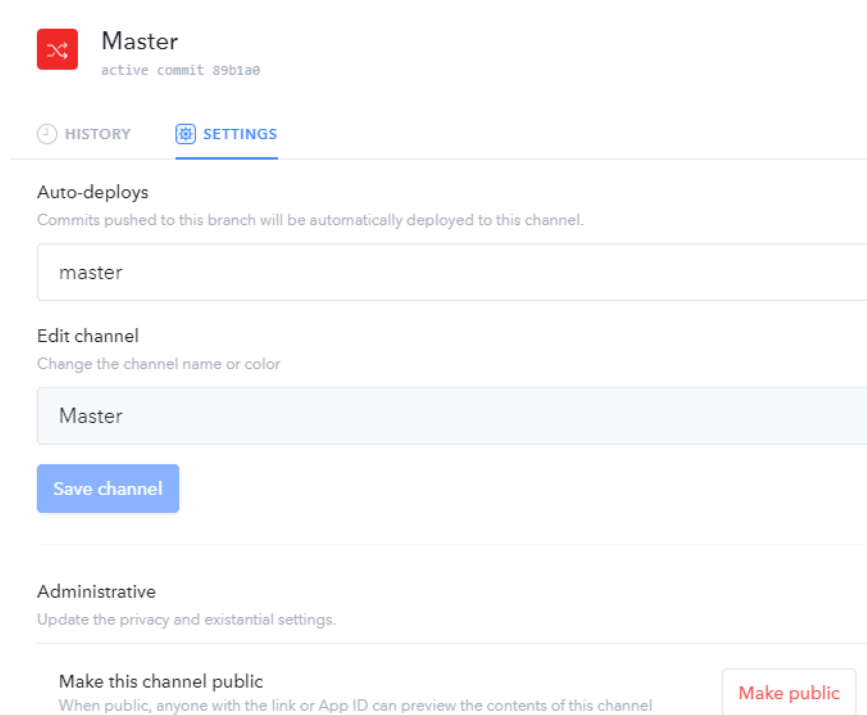


Figure 1-s: The Master Channel Settings in Ionic Pro

On your device, open the Ionic View application you previously installed, and go to the settings tab to enter your Ionic Pro credentials.

Figure 1-t shows how this looks from my device—it might look similar on yours.

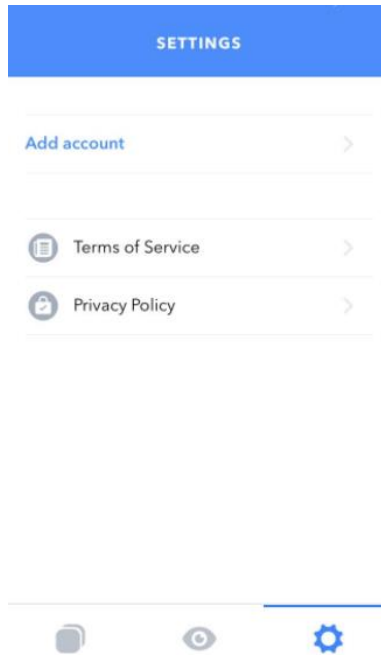


Figure 1-t: Settings Tab on the Ionic View App on My Device

Tap **Add Account** and enter your Ionic Pro email address and password. With this done, Ionic View is now linked to your Ionic Pro account. Next, tap the middle icon (which looks like an eye) and enter the app ID code.

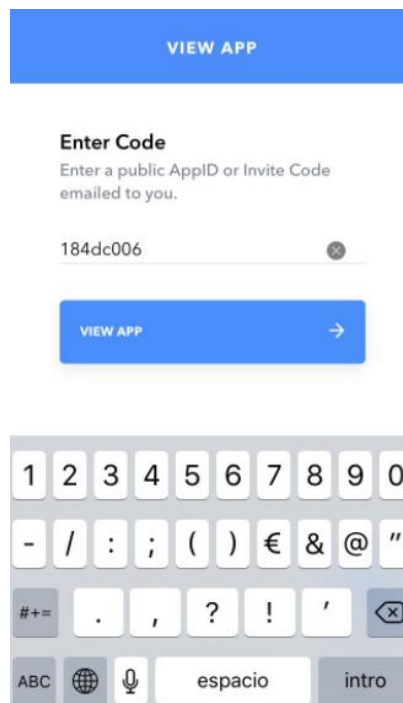
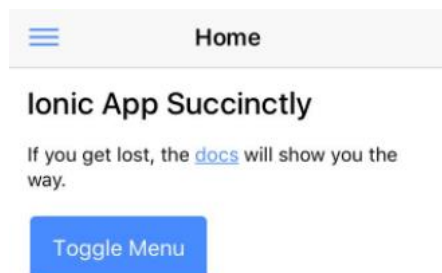


Figure 1-u: Entering the App ID on Ionic View on My Device

Once done, tap **VIEW APP** to see the app running on Ionic View. Once you've tapped the **VIEW APP** button, you should see the name **IonicAppSuccinctly** on the list—tap it in order to load the application on your device. Here is how it looks on my iPhone.



*Figure 1-v: The App Running on Ionic View*

We've quickly seen how we can use Ionic Pro and Ionic View to deploy and test our Ionic app, respectively.

If we have other apps to view within Ionic View, we can simply go back to the app list and exit the current application we are viewing by shaking our device when the Ionic View app is running—very handy!

We've just scratched the surface of what is possible with Ionic Pro and gone through the steps required to have our code deployed in the cloud so it can be tested with the Ionic View app.

Ionic Pro's main feature is that it makes it easy to push remote app updates in real-time without going through traditional App Store release processes.

## Summary

In this chapter, we jumped right into the Ionic ecosystem and scaffolded the basic structure of the application we'll be building throughout this book.

We also installed the necessary tooling and went through the steps required to test our application, both in the browser, and using the Ionic View app—which also introduced us to Ionic's cloud, Ionic Pro.

In the next chapter, we'll explore the scaffolded project structure in detail in order to understand it well and be able to expand the app's functionality later.

We'll skip how to test the app on an emulator, as this requires setting up various SDKs, which is a complicated process. However, I encourage you to check Ionic's official [documentation](#) on this subject.

Later in the upcoming chapters, we'll see how Ionic-Angular components work so we can start to build our scaffolded app from the ground up.

# Chapter 2 Project Structure

## Quick intro

In the previous chapter, we were able to create our Ionic app using a predefined template, which we will be using as the base for the solution we'll be working on throughout this book.

In order to make things fun, I thought of creating an app that would allow us to browse and search through the *Succinctly* series library, as well as organize the books into learning paths. So that's the app we'll be building—let's get started!

## The \src folder

Now that we've looked at how we can deploy and test our app on Ionic Pro, let's focus on the app structure and the code that was created for us by the Ionic CLI. So open up VS Code in order to explore the folder structure.

The \src folder is where we'll be spending most of our time when developing any Ionic-Angular application. Here's where any initialization code goes.

Within the \src folder is where all the application's source files are found, including images, styles, HTML markup, and code files (such as TypeScript or JavaScript). By default, the \src folder is organized as follows.

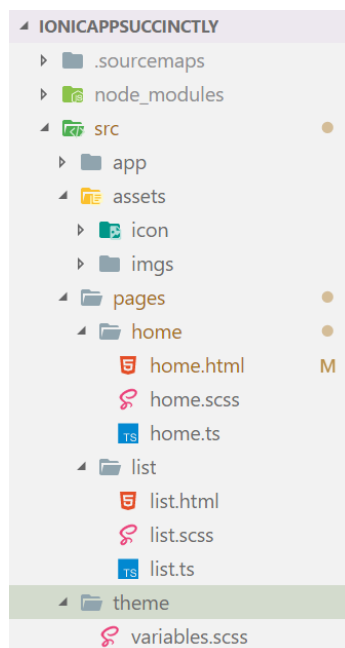


Figure 2-a: The App's \src Folder Structure

Let's quickly go over this structure in order to understand how things have been organized. The `\src` folder is split into various subfolders, each with a defined purpose.

## The `\src\app` folder

The `\src\app` folder contains the application's main module and component. Ionic applications are organized into modules that use one or more components.

Within the `\src\app` folder, `main.ts` is the entry point for our Ionic application—it is responsible for importing the app's main module, which corresponds to `app.module.ts`. We can see this in the following code listing.

*Code Listing 2-a: `main.ts`*

```
import { platformBrowserDynamic } from '@angular/platform-browser-  
dynamic';  
import { AppModule } from './app.module';  
platformBrowserDynamic().bootstrapModule(AppModule);
```

Within `app.module.ts`, there are references to the app's main component, `app.component.ts`, and to the two predefined pages that our application has—the **Home** and **List** pages. We can see this as follows.

*Code Listing 2-b: Component References in `app.module.ts`*

```
import { MyApp } from './app.component';  
import { HomePage } from '../pages/home/home';  
import { ListPage } from '../pages/list/list';
```

Here comes the interesting part—the app's main component, `app.component.ts`, is actually associated with two other files: `app.html` and `app.scss`, which correspond to the application's main view and style, respectively. The `app.scss` file is used for setting styling rules that apply to the application in general; you can find more information [here](#).

If we open **`app.component.ts`**, we can see that it references `app.html` as its view template—as follows.

*Code Listing 2-c: `app.component.ts` Referencing `app.html`*

```
@Component({  
  templateUrl: 'app.html'  
})
```

So you can think of the `\src\app` as the repository that contains the application's main component—and the module loads it, which references the rest of the app's parts.

## The \src\assets folder

This folder basically contains static assets, such as icons and images that the application will use. By default, it contains an \icon folder that includes the app's favicon and a \imgs folder that contains the app's logo. Here we can add any assets that our application will use.

## The \src\pages folder

As its name clearly implies, this folder contains the navigation pages our application actually uses.

Each navigation page has a separate subfolder. By default, the template that was created by the Ionic CLI has two pages: one called **home**, and the other called **list**—respectively found under the home and list folders.

Under each of these subfolders, each navigation page is actually made up of several files: an HTML (.html), a style sheet (.scss), a TypeScript component (.ts) code file, and most recently, a TypeScript (.module.ts) code file.

Let's have a look at home.ts—you should be able to see code similar to the following.

*Code Listing 2-d: home.ts*

```
import { Component } from '@angular/core';
import { IonicPage, NavController, NavParams } from 'ionic-angular';

@IonicPage()
@Component({
  selector: 'page-home',
  templateUrl: 'home.html'
})
export class HomePage {
  constructor(public navCtrl: NavController, public navParams: NavParams) {
  }

  ionViewDidLoad() {
    console.log('ionViewDidLoad HomePage');
  }
}
```

Notice that because Ionic-Angular is a very quickly evolving framework, the Ionic team is always incorporating the latest Angular framework changes into Ionic-Angular, so it is possible that between small updates to the Ionic CLI—when you scaffold a new project—the out-of-the-box code might look slightly different to what is listed in the previous code listing.



So it is possible that the scaffolded code for `home.ts` might instead look similar to the following one. In my case, the CLI has scaffolded the code for `home.ts` as follows.

*Code Listing 2-e: Another Possible Version of `home.ts`*

```
import { Component } from '@angular/core';
import { NavController } from 'ionic-angular';

@Component({
  selector: 'page-home',
  templateUrl: 'home.html'
})
export class HomePage {
  constructor(public navCtrl: NavController) {
  }
}
```

Ionic apps follow the [Model-view-controller](#) (MVC) design pattern. As we can see, a reference to `home.html` is found within `home.ts`. The HTML file represents the home page's view, while the TypeScript file (`home.ts`) is the controller, which contains the Angular component.

The Angular component is represented by the **HomePage** class, which has a constructor that receives an Ionic **NavController** object as a parameter. You can find more information about the **NavController** class [here](#).

The style sheet (`.scss`) file is specific to the view, and it is compiled down to CSS by Ionic when the application is built.

## The `\src\theme` folder

Next on our list is the `\src\theme` folder, which defines the Ionic theming used by the application. We don't need to touch this folder or the `variables.scss` file unless we want to use a different theme—you can find more information about this topic [here](#).

## The files within the `\src` folder

The last part of the `\src` folder that we need to talk about is regarding the files that reside in its root, which are: `index.html`, `manifest.json`, and `service-worker.js`.

As you would expect, the `index.html` file defines the runtime entry point to the Single Page Application ([SPA](#)) that contains the entire Ionic app. In essence, what Ionic does is create an SPA that is then transformed into a native mobile app when deployed to iOS or Android.

The reason there's a `manifest.json` file is that with Ionic, the apps we build are by default easily upgradable to [Progressive Web Apps](#) (PWAs).

This means that even if the app is not compiled into a mobile app for a specific platform, it can still be installed on the home screen of the mobile device as a PWA.

This requires that we use a [web manifest](#) file. Our manifest file looks like this.

*Code Listing 2-f: manifest.json*

```
{
  "name": "Ionic",
  "short_name": "Ionic",
  "start_url": "index.html",
  "display": "standalone",
  "icons": [{
    "src": "assets/imgs/logo.png",
    "sizes": "512x512",
    "type": "image/png"
  }],
  "background_color": "#4e8ef7",
  "theme_color": "#4e8ef7"
}
```

This manifest file is then tied up with the app's main runtime entry point, the index.html file, as follows.

*Code Listing 2-g: The index.html Referencing the manifest.json File*

```
<link rel="manifest" href="manifest.json">
```

The service-worker.js file is like any other JavaScript file, with the exception that it runs in the background and is triggered via events—this is what is known as a Service Worker.

Service Workers are incredibly powerful, and honestly, incredibly confusing to some extent. They provide the app's offline functionality, push notifications, background content updating, caching, and other background features. At a high level, they run behind the scenes, independently of the app, in response to various events like network requests, push notifications, and connectivity changes.

The service-worker.js file created by the Ionic CLI provides a basic structure for how to organize these types of tasks—you can find more information about this topic [here](#).

If you would like to better understand Ionic's philosophy for enabling PWAs out of the box, I recommend you read this excellent [blog post](#), which explains it all.

## The `\node_modules` folder

This folder contains the Node.js packages that the application uses—which were installed when we ran the `npm install` command. These packages are specified within the `package.json` file found on the app's root folder.

## The `\www` folder

This folder is where Ionic will deploy the application once it has been transpiled and built.

## The `\` (root) folder

The app's root folder contains some very important files, without which the application wouldn't be able to do much.

The `package.json` file contains the name of the Node.js modules that the app uses, whereas files such as `tsconfig.json` and `tslint.json` contain general TypeScript configuration and linting settings, respectively. Linting is the process of checking TypeScript or JavaScript code for readability, maintainability, and functionality errors.

Other files in the root folder, such as `.gitignore`, `.editorconfig`, and `ionic.config.json`, provide essential configuration settings that are specific for Git, editing, and Ionic, respectively.

Finally, the `package-lock.json` file is automatically generated for any operations where the Node Package Manager (NPM) modifies either the `\node_modules` folder, or the `package.json` file.

The purpose of the `package-lock.json` file is to describe the exact folder tree that was generated, so that subsequent installs are able to generate identical trees, regardless of intermediate dependency updates. You can find more information [here](#).

## Summary

With the project structure covered, we are now ready to start exploring Ionic's navigation system, and later, its components—which are both essential in order to start making the necessary modifications to our application.

Sounds exciting! So let's not wait any longer, and start exploring Ionic's navigation system in order to kickstart our app.

# Chapter 3 Starting the App

## Quick intro

We've managed to cover a bit of ground—we've gotten Ionic up and running, and also explored the project structure that the Ionic CLI created for us.

However, in order to be able to expand our application and give it the functionality we want to achieve, we'll need to understand how Ionic's navigation system works. This will be the focus of this chapter. With this said, let's jump right in and begin exploring Ionic's fascinating and well-designed navigation system.

## Overview

Navigation in Ionic-Angular is completely different from how it was in Ionic 1, as it has been rewritten from the ground up.

In Ionic-Angular, the foundation of the navigation system is known as the navigation stack—which is what we'll need to understand first. Understanding how the navigation stack works will help us modify our app and put the necessary views in place that our application will eventually use.

Like with any navigation system, we'll also explore how to pass parameters between views.

## Navigation stack

As the name implies, Ionic's navigation system is based on the concept of a stack, which means that views are mounted one on top of the other.

This stack provides an API for navigation that Ionic-Angular apps use; however, Ionic is doing a bit more than just providing us a stack to mount our app's views. Behind the scenes, Ionic helps improve our app's performance for us by caching pages in the DOM.

When a user triggers an action on our app that opens a view, a new page is pushed onto the stack in forward navigation, and then popped off the stack when navigating backward. In essence, this is the typical mechanism of how a navigation stack works.

Nowadays, many applications will use just a single navigation stack, known as the root stack. Nevertheless, Ionic supports multiple navigation stacks—a tabbed layout contains its own navigation stack, which is totally independent of the root stack.

Figure 3-a is an illustration of how a typical navigation stack would look to the naked eye, if we could actually see how views are internally organized.



*Figure 3-a: A Typical Navigation Stack*

Notice how pages stack up one on top of the other. The most recent page (in purple) sits at the top, while the first one opened by the user (in blue) is right at the bottom of the stack.

## Main difference from Ionic for AngularJS

In Ionic 1 (with AngularJS), the navigation system was based on the UI Router component, which revolved around the usage of URLs for navigation. Despite URLs not being fully required in Ionic 1's navigation system, as named routes could also be used, Ionic-Angular completely departed from this. The navigation works based on a stack, so URLs don't matter at all any more, at least for basic navigation.

Ionic-Angular still supports URLs, but for deep linking, which is the capability of allowing a user to jump straight into a view contained deeper inside the app, such as one that could be launched from a web browser.

## App navigation overview

In order to better understand the navigation system of the app we will be building, it's best to sketch it out in a simple diagram.

In Figure 3-b, the app's main page will consist of a list of the books present in the *Succinctly* series. From there, there will be two possible routes to navigate—either to the overview of a specific book (and its details), or to the learning path associated to that book, which will include a list of related books.

From the list of related books, it will be also possible to navigate directly to the overview of a specific book and to its details.

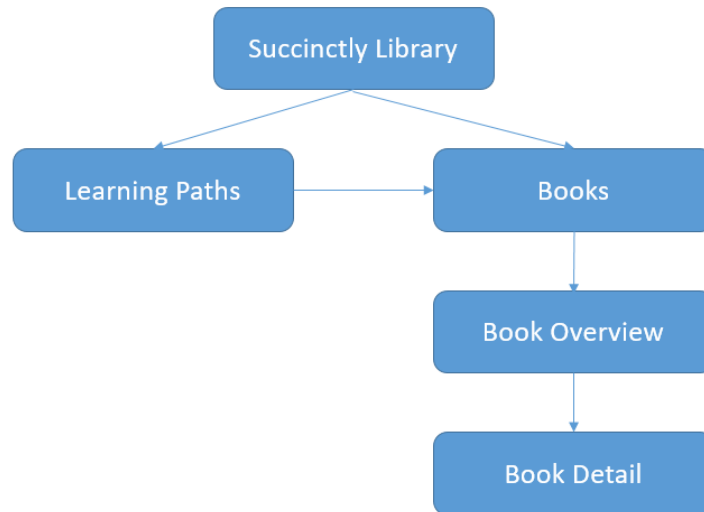


Figure 3-b: Our App's Pages

Before we even start to implement this navigation system, we first need to generate several pages for these screens.

Let's start with the **library** page, since this is going to serve as the home page of our app.

## Creating the app's main page

From here on, we'll start to write code. I usually prefer to use lighter themes when writing code (rather than darker ones)—that's just a personal preference, so I'll change my Visual Studio Code theme to a lighter one, which you'll notice on the screen shots that will follow.

To create the app's main page, navigate in Visual Studio Code to the `\src\pages` folder and create a new folder under it, called **library**.

This will be the main page for our application, which corresponds to the **Succinctly Library** page from Figure 3-b.

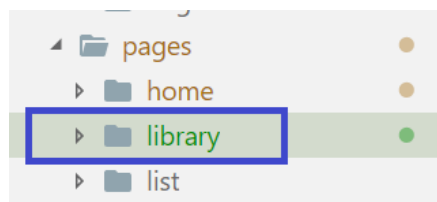


Figure 3-c: The "library" Folder

With this folder created, let's create a TypeScript file in the new directory that is going to serve as our component. We'll call this file **library.ts**.

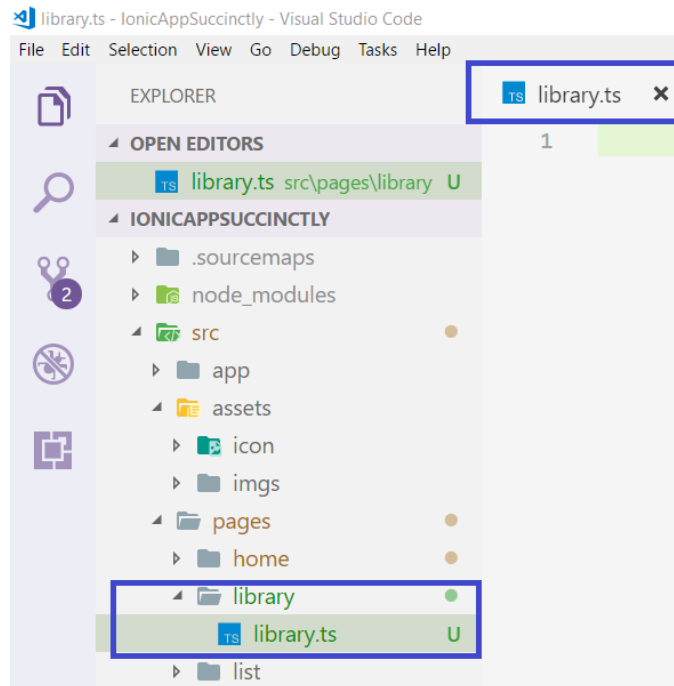


Figure 3-d: The library.ts File

After creating this file, let's go ahead and add the following boilerplate code. This will be the base for this component.

Code Listing 3-a: Boilerplate Code for library.ts

```
import { Component } from '@angular/core';
import { IonicPage, NavController, NavParams } from 'ionic-angular';

@IonicPage()
@Component({
  templateUrl: 'library.html',
})
export class LibraryPage {

  constructor(public navCtrl: NavController, public navParams: NavParams)
  { }

  ionViewDidLoad() {
    console.log('ionViewDidLoad LibraryPage');
  }
}
```

This code simply creates a TypeScript Angular component that references an HTML file, which will contain the layout for this page. We'll add extra features as we go.

Creating this component also requires we create the corresponding HTML file for it, which we'll call **library.html**. We can create this file in the same **library** folder.

Once you have created this file, please add the following boilerplate code to it.

*Code Listing 3-b: Boilerplate Code for library.html*

```
<ion-header>
  <ion-nav-bar>
    <ion-title>Succinctly Library</ion-title>
  </ion-nav-bar>
</ion-header>

<ion-content padding>
</ion-content>
```

Looking at this code, we can see that we have an **ion-header** with an **ion-nav-bar** component, in which we have placed the page's title.

It is suggested that Ionic-Angular pages have a corresponding **NgModule**. This means we should manually also create a **library.module.ts** file; however, using the syntax described in Code Listing 2-e eliminates the need to have a **.module.ts** file for every page.

There's a way to automatically generate a **.module.ts** file (and all the files corresponding to a new page). This can be achieved by running the **ionic generate page »page name«** command.

For now, let's manually create a **library.module.ts** file in the **library** folder with the following boilerplate code.

*Code Listing 3-c: Boilerplate Code for library.module.ts*

```
import { NgModule } from '@angular/core';
import { IonicPageModule } from 'ionic-angular';
import { LibraryPage } from './library';

@NgModule({
  declarations: [
    LibraryPage,
  ],
  imports: [
    IonicPageModule.forChild(LibraryPage),
  ],
})
export class LibraryPageModule {}
```



We've now created the basic code templates for our main page. Before we create other pages, let's quickly have a look at how to make our Ionic-Angular development life easier.

## Visual Studio Code Ionic plugins

Visual Studio Code is a great development environment, and as such, it has a vast ecosystem of third-party plugins that make a developer's life much easier.

To make my Ionic-Angular development easier, I've installed some plugins through the **Extension** option by typing in **ionic** or **ionic2**, as seen in the following figure.

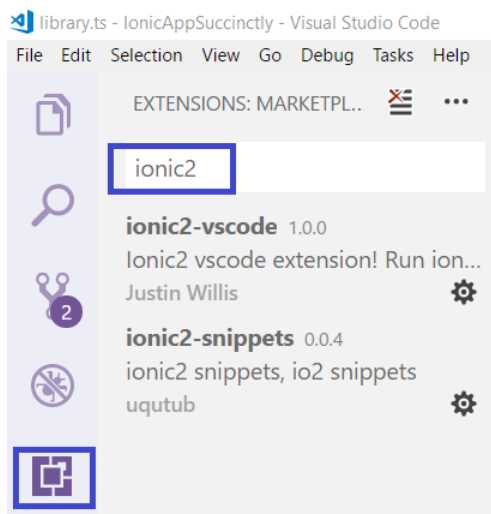


Figure 3-e: Ionic Extensions for Visual Studio Code

Once the plugins are installed, you'll see a small button next to each of the ones you installed with a request to **Reload** Visual Studio Code.

To work with the plugins, click any of the **Reload** buttons—this will close the current running instance of Visual Studio Code and open up a new one.

We now have some cool Ionic plugins installed that we can work with.

## Creating pages with the CLI

I'm a bit of an old-school person, which means I like to actually write code from scratch. Nowadays, however, many tools—including the Ionic CLI—are able to generate boilerplate code by simply running a command. Let me show you how you can do this with Ionic-Angular.

I'll open the command prompt and navigate to the folder where I have the code for the app we are building. All I need to do in order to generate a new page is enter the following command.

Code Listing 3-d: Ionic Generate Page Command

```
ionic generate page paths
```

The keyword **paths** is the actual name of the directory that will be generated by the Ionic CLI. Once the command has successfully executed, we'll see an output similar to this.

```
λ ionic generate page paths  
[OK] Generated a page named paths!
```

Figure 3-f: Ionic CLI Command Successfully Executed

How cool is that? If we now inspect the folder structure within Visual Studio Code, we can see that the following files were added.

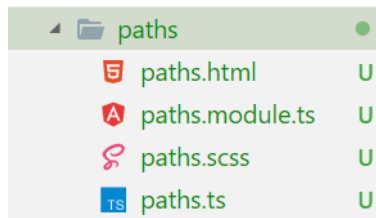


Figure 3-g: The “Paths” Page Files

The **paths** files correspond to the **Learning Paths** page from Figure 3-b. With this one out of the way, let's create the remaining pages for our app with the Ionic CLI.

Let's create the **Books** page—we can do this by executing the following command.

Code Listing 3-e: Generate the Books Page

```
ionic generate page books
```

The next one is the **Book Overview** page. We can generate it by running the following command.

Code Listing 3-f: Generate the Book Overview Page

```
ionic generate page bookoverview
```

The last page we need to generate for our app is the **Book Detail** page. So let's create it with the following command.

Code Listing 3-g: Generate the Book Details Page

```
ionic generate page bookdetail
```

If you like, you can also automatically generate the **Library** page files by running the following command. But before you do that, remove the **library.\*** files you created previously.

*Code Listing 3-h: Generate the Library Page*

```
ionic generate page library
```

All the pages that our application will use have now been generated. We can inspect their files by going into their respective folders in Visual Studio Code.

## Wiring up the new pages

Now that we have all the pages our app will need, we need to remove the references to the two pages that came out of the box, and that the CLI generated for us when the application was created: **Home** and **List**.

In order to do this, open the **app.component.ts** file under the **\src\lapp** folder, remove the references to the **home** and **list** pages, and add the references to some of the pages we've created, such as: **Library**, **Paths**, and **Books**.

We also need to point the **rootPage** variable to **LibraryPage** instead of **HomePage**. Furthermore, **this.pages** (which is used for the side menu) has also been updated.

The updated **app.component.ts** file should now look as follows.

*Code Listing 3-i: The Updated app.component.ts File*

```
import { Component, ViewChild } from '@angular/core';
import { Nav, Platform } from 'ionic-angular';
import { StatusBar } from '@ionic-native/status-bar';
import { SplashScreen } from '@ionic-native/splash-screen';

import { LibraryPage } from '../pages/library/library';
import { PathsPage } from '../pages/paths/paths';
import { BooksPage } from '../pages/books/books';

@Component({
  templateUrl: 'app.html'
})
export class MyApp {
  @ViewChild(Nav) nav: Nav;

  rootPage: any = LibraryPage;

  pages: Array<{title: string, component: any}>;
```

```

constructor(public platform: Platform, public statusBar: StatusBar,
  public splashScreen: SplashScreen) {
  this.initializeApp();

  this.pages = [
    { title: 'Paths', component: PathsPage },
    { title: 'Books', component: BooksPage }
  ];
}

initializeApp() {
  this.platform.ready().then(() => {
    this.statusBar.styleDefault();
    this.splashScreen.hide();
  });
}

openPage(page) {
  this.nav.setRoot(page.component);
}
}

```

Now, run the **ionic serve** command and open your browser to see how it looks. In my case, I see the following.

## Error

### Runtime Error

Uncaught (in promise): Error: No component factory found for LibraryPage. Did you add it to @NgModule.entryComponents?  
 noComponentFactoryError (http://localhost:8100/build/vendor.js:4142:34) at CodegenComponentFactoryResolver.resolveCc  
 (http://localhost:8100/build/vendor.js:21974:20) at NavControllerBase.\_viewInit (http://localhost:8100/build/vendor.js:4872:  
 (http://localhost:8100/build/polyfills.js:3:14976) at Object.onInvoke (http://localhost:8100/build/vendor.js:4982:33) at t.invo  
 (http://localhost:8100/build/polyfills.js:3:10143) at http://localhost:8100/build/polyfills.js:3:20242

### Stack

Error: Uncaught (in promise): Error: No component factory found for LibraryPage. Did you add it to @NgModule.entryComponents?  
 Error: No component factory found for LibraryPage. Did you add it to @NgModule.entryComponents?  
 at noComponentFactoryError (http://localhost:8100/build/vendor.js:4142:34)  
 at CodegenComponentFactoryResolver.resolveComponentFactory (http://localhost:8100/build/vendor.js:4206:19)

*Figure 3-h: A Runtime Error*

Oops! Wait, what happened here?

We are getting this error because we also need to edit the `app.module.ts` file, remove the old references to the **Home** and **List** pages, and add the references to the **Library** page.

Here's what the updated app.module.ts file looks like.

*Code Listing 3-j: The Updated app.module.ts File*

```
import { BrowserModule } from '@angular/platform-browser';
import { ErrorHandler, NgModule } from '@angular/core';
import { IonicApp, IonicErrorHandler, IonicModule } from 'ionic-angular';

import { MyApp } from './app.component';
import { LibraryPage } from '../pages/library/library';

import { StatusBar } from '@ionic-native/status-bar';
import { SplashScreen } from '@ionic-native/splash-screen';

@NgModule({
  declarations: [
    MyApp,
    LibraryPage
  ],
  imports: [
    BrowserModule,
    IonicModule.forRoot(MyApp),
  ],
  bootstrap: [IonicApp],
  entryComponents: [
    MyApp,
    LibraryPage
  ],
  providers: [
    StatusBar,
    SplashScreen,
    {provide: ErrorHandler, useClass: IonicErrorHandler}
  ]
})
export class AppModule {}
```

What I've done is update the **NgModules** **declarations** and **entryComponents** arrays with the **Library** page name.

Because we are running the **ionic serve** command, it is performing live, reloading when any changes take place. Let's now have a look at the browser.

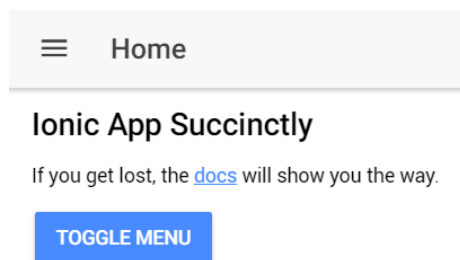


*Figure 3-i: The App's Updated UI (1)*

The **Library** page is now set as our app's home page, based on the changes we've recently made.

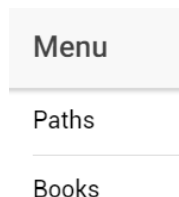
However, as you can appreciate, there's not much of a user interface yet. To speed things up a bit, simply open up the **home.html** file, copy its content, and replace the content of **library.html** with the content you copied.

Have a look at your browser—now the UI should look as follows.



*Figure 3-i: The App's Updated UI (2)*

Given that we previously updated **this.pages** on `app.component.ts` with the names of the main pages we created with the CLI—specifically, **Paths** and **Books**—the side menu should now look as follows.



*Figure 3-j: The Updated Side Menu*

We now have all our new pages wired up to the original scaffolded project we created using the CLI.

## Summary

In this chapter, we've quickly looked at how the navigation stack in Ionic-Angular works and started off our project by adding the pages we'll need in order to build our app.

We also wired these pages up to the project that was originally scaffolded by adding the necessary references to the `app.component.ts` and `app.module.ts` files.

We are now ready to actually build our app—this is what we'll do in the next chapter. Let's get our hands dirty.

# Chapter 4 Building the App

## Quick intro

One of the things I love most about Ionic is how easy it is to cover a lot of ground with barely any information overload. So far, we've seen how to scaffold a project, add pages, and wire those pages up, and also how to do testing with Ionic View and the browser, and deploy to Ionic Pro.

That's actually an impressive feat on its own, considering that it's all been outlined in a small number of pages. This is only possible because the framework accommodates for this level of brevity without sacrificing on essential content.

But like everything in life, the best way to learn something new—no matter how concise the content might be—is by building something. That's what we'll be doing throughout this chapter.

To make the most of this chapter, I highly advise you to check the great Ionic component documentation as we go along, also, which you can find [here](#).

At the end of this chapter, you can find the link to download the full source code of this app.

## Before we start

But before we start, I want to mention that I like to keep things simple—because it not only allows for easier knowledge retention, but also limits the scope to just the necessary essentials required to get something done.

With this philosophy in mind, let's create our **Succinctly Library** app by keeping it as simple as possible. This means our app won't use a database or perform AJAX requests, so all the data required will be hard-coded.

Of course, you may later modify the app to make it more complex and include such features, but that's something I'll totally leave up to you.

## The Library page

The app's **Library** page will be the main entry point to the application. As we've recently seen, it contains the side menu with the links to the other main pages of the app.

I'm a big fan of having a side menu in an app; however, I believe that the main content of the **Library** page should include some cards that also point or reference the other main pages of the app, such as **Books** and **Paths**—so let's add some cards to the **Library** page in order to achieve this.

In order to add some data that we'll display on these cards, we'll need to open up the **library.ts** file and add the following code.

*Code Listing 4-a: Card Data on the library.ts File*

```
cardData = [
  {title: 'Paths', description: 'Learning paths',
   pushPage: 'PathsPage'},
  {title: 'Books', description: 'Books in the library',
   pushPage: 'BooksPage'}
];
```

This code can be added to the **LibraryPage** class just before the **constructor**. This is how the updated library.ts file with this code now looks:

*Code Listing 4-b: Updated library.ts File*

```
import { Component } from '@angular/core';
import { IonicPage, NavController, NavParams } from 'ionic-angular';
import { PathsPage } from '../pages/paths/paths';
import { BooksPage } from '../pages/books/books';

@IonicPage()
@Component({
  selector: 'page-library',
  templateUrl: 'library.html',
})
export class LibraryPage {
  cardData = [
    {title: 'Paths', description: 'Learning paths',
     pushPage: 'PathsPage'},
    {title: 'Books', description: 'Books in the library',
     pushPage: 'BooksPage'}
  ];

  constructor(public navCtrl: NavController, public navParams: NavParams) {
  }

  ionViewDidLoad() {
    console.log('ionViewDidLoad LibraryPage');
  }
}
```

We'll use the **cardData** object within the library.html file in order to build a couple of cards to display within the UI. Let's do this next.



Let's open up **library.html** and add the following code within the **ion-content** tag.

*Code Listing 4-c: The ion-card Markup*

```
<ion-list *ngFor="let card of cardData" no-padding>
  <ion-card>
    <div class="card-title"
      [navPush]=card.pushPage>{{card.title}}
    </div>
  </ion-card>
</ion-list>
```

Let's quickly examine this. We've added the **ion-list** component in order to be able to iterate through the various elements of the **cardData** object—this can be achieved by using **ngFor**.

Each element of **cardData**, represented by the **card** object, is then referenced within the **ion-card** component. The **card.pushPage** property is used to allow the user to navigate to the page that corresponds to **card.title**.

After making this change, this is how the **library.html** file looks:

*Code Listing 4-d: The Updated library.html File*

```
<ion-header>
  <ion-navbar>
    <button ion-button menuToggle>
      <ion-icon name="menu"></ion-icon>
    </button>
    <ion-title>Succinctly Library</ion-title>
  </ion-navbar>
</ion-header>

<ion-content padding>
  <h3>Welcome to the Succinctly Series</h3>
  <ion-list *ngFor="let card of cardData" no-padding>
    <ion-card>
      <div class="card-title"
        [navPush]=card.pushPage>{{card.title}}</div>
    </ion-card>
  </ion-list>
  <button ion-button secondary menuToggle>Toggle Menu</button>
</ion-content>
```

We can preview the UI changes in the browser by running the **ionic serve** command.

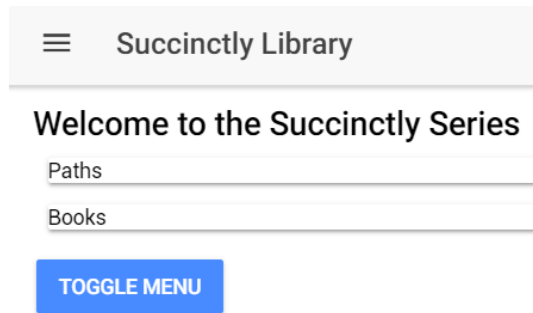


Figure 4-a: The Updated Library Page

In order to test Ionic-Angular’s navigation features, let’s click on either the **Paths** or the **Books** card. As expected, and as seen in Figure 4-b, the corresponding page is displayed and added to the navigation stack. This is possible because `card.pushPage` references the page that Ionic will mount.

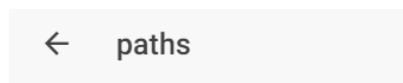


Figure 4-b: The Mounted Paths Page

We can unmount the current page on the navigation stack and return to the previous one—in this case, the **Library** page—by simply clicking the back arrow button.

We now have our **Library** page working; however, we can still improve the way the card component looks, so let’s do that next.

## Making a nicer card

As the saying goes, “looks are not everything, but they do matter.” In today’s highly competitive app market, users are spoiled with choice, and nice-looking, appealing apps have a better chance of remaining on people’s devices than ones that have less appealing UIs.

Therefore, it is somewhat important to create a good user experience, and in order to do that, having a relatively appealing UI is key.

As the card component will be an essential part of our application, and probably used on most of the app’s pages, it’s worthwhile to spend some time creating a nicer-looking card. This will not only make our app look better, but will also provide the user a better experience.

Let’s go ahead and do that. The following code will be the base for our new and improved card.

Code Listing 4-e: A Nicer Card

```
<ion-card>
  <ion-item>
```

```

    <h2>Card Header</h2>
    <p>Card Short Description</p>
</ion-item>
<img src="" />
<ion-card-content>
    <ion-card-title>
        Card Title
    </ion-card-title>
    <p>
        Card Long Description
    </p>
</ion-card-content>
<ion-row>
    <button ion-button [navPush]=card.pushPage>{{card.title}}</button>
</ion-row>
</ion-card>

```

Let's now take this code and replace the previous card within the **library.html** file. The library.html file will now look as follows.

*Code Listing 4-f: The Updated library.html File*

```

<ion-header>
    <ion-navbar>
        <button ion-button menuToggle>
            <ion-icon name="menu"></ion-icon>
        </button>
        <ion-title>Succinctly Library</ion-title>
    </ion-navbar>
</ion-header>

<ion-content padding>
    <h3>Welcome to the Succinctly Series</h3>
    <ion-list *ngFor="let card of cardData" no-padding>
        <ion-card>
            <ion-item>
                <h2>Card Header</h2>
                <p>Card Short Description</p>
            </ion-item>
            <img src="" />
            <ion-card-content>
                <ion-card-title>
                    Card Title
                </ion-card-title>
            </ion-card-content>
        </ion-card>
    </ion-list>
</ion-content>

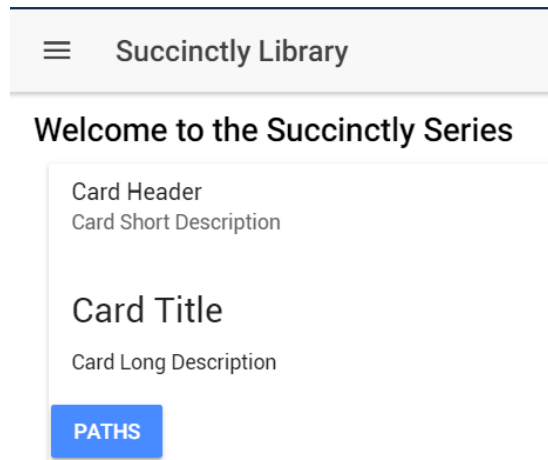
```

```

        <p>
            Card Long Description
        </p>
    </ion-card-content>
</ion-row>
    <button ion-button
        [navPush]=card.pushPage>{{card.title}}
    </button>
</ion-row>
</ion-card>
</ion-list>
<button ion-button secondary menuToggle>Toggle Menu</button>
</ion-content>

```

Let's now execute the **ionic serve** command to see how this looks.



*Figure 4-c: The Updated Library Page*

That looks a bit nicer than the previous version of the card, as it contains an image (which we still need to specify), various text elements, and a button.

Not super sexy yet from a UI point of view, but at least something that provides a bit more information and a better experience to the user.

With this new card structure and outline, let's add the card elements required for the **Paths** and **Books** pages and upgrade the look and feel of the **Library** page.

## Updating the Library page

Now that we've got a new card style, let's finalize updating the **Library** page. From a UI perspective, the HTML markup of the **Library** page is pretty much done; however, we need make those text elements dynamic for each card.

Let's start doing this by expanding the **cardData** object within the **library.ts** file and replacing it with the code that follows.

*Code Listing 4-g: The Updated cardData Object*

```
cardData = [
  { img: '../assets/imgs/logo.png',
    header: 'Paths',
    short: 'Learning paths',
    title: '',
    description: 'Paths are books organized by related topics',
    buttontxt: 'Explore Paths',
    pushPage: 'PathsPage'
  },
  { img: '../assets/imgs/azure.jpg',
    header: 'Books',
    short: 'Books to read',
    title: '',
    description: 'Over 130 books on technologies that matter',
    buttontxt: 'Browse Books',
    pushPage: 'BooksPage'
  }
];
```

As you may have noticed in Code Listing 4-g, I've also dropped a couple of images into the `\assets\imgs\` folder. So, after updating the **cardData** object, our **library.ts** file now looks as follows.

*Code Listing 4-h: The Updated library.ts File*

```
import { Component } from '@angular/core';
import { IonicPage, NavController, NavParams } from 'ionic-angular';
import { PathsPage } from '../pages/paths/paths';
import { BooksPage } from '../pages/books/books';

@IonicPage()
@Component({
  selector: 'page-library',
  templateUrl: 'library.html',
})
export class LibraryPage {
  cardData = [
    { img: '../assets/imgs/logo.png',
      header: 'Paths',
      short: 'Learning paths',
```

```

        title: '',
        description: 'Paths are books organized by related topics',
        buttontxt: 'Explore Paths',
        pushPage: 'PathsPage'
    },
    {
        img: '../../../assets/imgs/azure.jpg',
        header: 'Books',
        short: 'Books to read',
        title: '',
        description: 'Over 130 books on technologies that matter',
        buttontxt: 'Browse Books',
        pushPage: 'BooksPage'
    }
];

constructor(public navCtrl: NavController, public navParams: NavParams)
{ }

ionViewDidLoad() {
    console.log('ionViewDidLoad LibraryPage');
}
}

```

The updated markup, library.html, now looks as follows.

*Code Listing 4-i: The Updated library.html File*

```

<ion-header>
  <ion-navbar>
    <button ion-button menuToggle>
      <ion-icon name="menu"></ion-icon>
    </button>
    <ion-title>Succinctly Library</ion-title>
  </ion-navbar>
</ion-header>
<ion-content padding>
  <h3>Welcome to the Succinctly Series</h3>
  <ion-list *ngFor="let card of cardData" no-padding>
    <ion-card>
      <ion-item>
        <h2>{{card.header}}</h2>
        <p>{{card.short}}</p>
      </ion-item>
      
    </ion-card>
  </ion-list>
</ion-content>

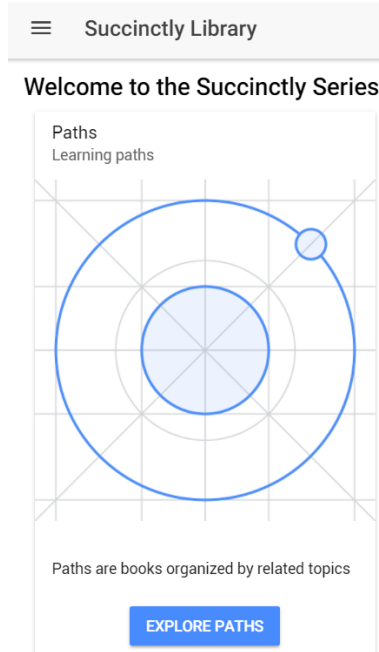
```

```

<ion-card-content>
  <ion-card-title>{{card.title}}</ion-card-title>
  <p>{{card.description}}</p>
</ion-card-content>
<ion-row center>
  <ion-col text-center>
    <button ion-button [navPush]=card.pushPage>
      {{card.buttontxt}}
    </button>
  </ion-col>
</ion-row>
</ion-card>
</ion-list>
</ion-content>

```

Let's execute the **ionic serve** command again in order to see how these changes look.



*Figure 4-d: The Updated Library Page*

Awesome! Our app looks much better than it did—small changes can go a long way.

With our **Library** page out of the way, let's focus on updating our **Paths** page.

## Updating the Paths page

As mentioned earlier, we are going to be hardcoding the data that our application will be using. The **Paths** page is nothing more than a small collection of topics into which books are organized—topics that are related to each other, thus forming a learning path.

In order to organize the book topics into paths, we need to create a **pathData** object within the **paths.ts** file that will indicate which book topics correspond to which learning path.

Let's define the **pathData** object as follows.

*Code Listing 4-j: The pathData Object*

```
pathData = [
  { name: 'Database',
    books: [
      { title: 'Force.com Succinctly', page: this.pg },
      { title: 'Azure Cosmos DB and DocumentDB Succinctly',
        page: this.pg }
    ]
  },
  { name: 'Web Development',
    books: [
      { title: 'Force.com Succinctly', page: this.pg }
    ]
  },
  { name: 'Cloud',
    books: [
      { title: 'Azure Cosmos DB and DocumentDB Succinctly',
        page: this.pg }
    ]
  },
  { name: 'Microsoft & .NET',
    books: [
      { title: 'Microsoft Bot Framework Succinctly', page: this.pg },
      { title: 'Twilio with C# Succinctly', page: this.pg },
      { title: 'Customer Success for C# Developers Succinctly',
        page: this.pg },
      { title: 'Data Capture and Extraction with C# Succinctly',
        page: this.pg },
    ]
  },
  { name: 'Mobile Development',
    books: [
      { title: 'Twilio with C# Succinctly', page: this.pg }
    ]
  }
]
```



```

    },
    { name: 'Miscellaneous',
      books: [
        { title: 'Customer Success for C# Developers Succinctly',
          page: this.pg }
      ]
    }
  ]
};

```

What we've done here is created a number of categories (learning paths), and under each, defined a number of books that correspond to them. Each book points to the **Book Overview** page—this is defined with **this.pg**.

To keep things simple—given that there are over 130 books in the series—I've just added the titles I've written, that have been published at the time of writing of this book.

With the **pathData** JSON data object defined, our **paths.ts** file now looks as follows.

*Code Listing 4-k: The Updated paths.ts File*

```

import { Component } from '@angular/core';
import { IonicPage, NavController, NavParams } from 'ionic-angular';

@IonicPage()
@Component({
  selector: 'page-paths',
  templateUrl: 'paths.html',
})
export class PathsPage {
  pg = 'BookoverviewPage';
  pathData = [
    { name: 'Database',
      books: [
        { title: 'Force.com Succinctly', page: this.pg },
        { title: 'Azure Cosmos DB and DocumentDB Succinctly',
          page: this.pg }
      ]
    },
    { name: 'Web Development',
      books: [
        { title: 'Force.com Succinctly', page: this.pg }
      ]
    },
    { name: 'Cloud',
      books: [

```

```

        { title: 'Azure Cosmos DB and DocumentDB Succinctly',
          page: this.pg }
      ],
    },
    { name: 'Microsoft & .NET',
      books: [
        { title: 'Microsoft Bot Framework Succinctly', page: this.pg },
        { title: 'Twilio with C# Succinctly', page: this.pg },
        { title: 'Customer Success for C# Developers Succinctly',
          page: this.pg },
        { title: 'Data Capture and Extraction with C# Succinctly',
          page: this.pg },
      ]
    },
    { name: 'Mobile Development',
      books: [
        { title: 'Twilio with C# Succinctly', page: this.pg }
      ]
    },
    { name: 'Miscellaneous',
      books: [
        { title: 'Customer Success for C# Developers Succinctly',
          page: this.pg }
      ]
    }
  ];

  constructor(public navCtrl: NavController, public navParams: NavParams)
  { }

  ionViewDidLoad() {
    console.log('ionViewDidLoad PathsPage');
  }
}

```

Let's now work on updating the UI of the **Paths** page. In order to do that, open up the **paths.html** file. Let's replace the existing markup with the following.

*Code Listing 4-1: The Updated paths.html File*

```

<ion-header>
  <ion-navbar>
    <ion-title>Paths</ion-title>
  </ion-navbar>

```

```

</ion-header>
<ion-content padding>
  <ion-list *ngFor="let path of pathData" no-padding>
    <ion-card>
      <ion-item>
        <ion-row center>
          <ion-col text-center>
            <h1><b>{{path.name}}</b></h1>
          </ion-col>
        </ion-row>
      </ion-item>
      <ion-list *ngFor="let book of path.books" no-padding>
        <ion-card-content>
          <ion-row center>
            <ion-col text-center>
              <h2>{{book.title}}</h2>
            </ion-col>
          </ion-row>
          <ion-row center>
            <ion-col text-center>
              <button ion-button [navPush]=book.page>
                Check book
              </button>
            </ion-col>
          </ion-row>
        </ion-card-content>
      </ion-list>
    </ion-card>
  </ion-list>
</ion-content>

```

As we can see, I've added some markup to the `paths.html` file so it looks relatively nice and is still simple enough to be easily read and understood.

I've added two **ngFor** loops—one that loops over the learning paths, and another that loops through the book titles of each learning path.

The rest of the code is simply regular markup that uses common standard Ionic components such as **ion-card**, **ion-item**, **ion-card-content**, **ion-row**, and **ion-col**, among regular HTML markup.

If we now execute the **ionic serve** command, our **Paths** page looks as follows.

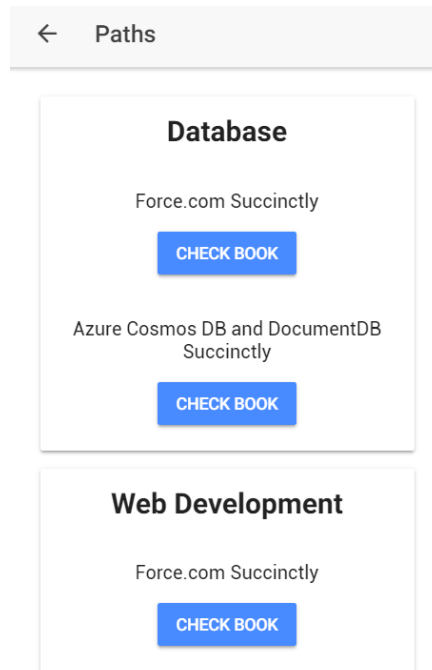


Figure 4-e: The Updated Paths Page

With a little code, we now have a good-looking page and the books organized by learning paths.

## Updating the Books page

Let's now move on and update the **Books** page, which will be quite similar to the **Paths** page. The difference is that the **Books** page will only contain information about the books within the library, without specifying which learning path they relate to.

You can think of the **Books** page as a trimmed-down version of the **Paths** path—instead of using two instances of **ngFor** in our markup, we'll require only one to loop around all the book titles.

First of all, let's define a **booksData** object within the **books.ts** file as follows.

Code Listing 4-m: The booksData Object

```
booksData = [
  { title: 'Force.com Succinctly', page: this.pg },
  { title: 'Azure Cosmos DB and DocumentDB Succinctly',
    page: this.pg },
  { title: 'Microsoft Bot Framework Succinctly', page: this.pg },
  { title: 'Twilio with C# Succinctly', page: this.pg },
  { title: 'Customer Success for C# Developers Succinctly',
    page: this.pg },
```

```

    { title: 'Data Capture and Extraction with C# Succinctly',
      page: this.pg }
  ];

```

With the **booksData** object defined, let's now update the **books.ts** file accordingly.

*Code Listing 4-n: The Updated books.ts File*

```

import { Component } from '@angular/core';
import { IonicPage, NavController, NavParams } from 'ionic-angular';

@IonicPage()
@Component({
  selector: 'page-books',
  templateUrl: 'books.html',
})
export class BooksPage {
  pg = 'BookoverviewPage';
  booksData = [
    { title: 'Force.com Succinctly', page: this.pg },
    { title: 'Azure Cosmos DB and DocumentDB Succinctly',
      page: this.pg },
    { title: 'Microsoft Bot Framework Succinctly', page: this.pg },
    { title: 'Twilio with C# Succinctly', page: this.pg },
    { title: 'Customer Success for C# Developers Succinctly',
      page: this.pg },
    { title: 'Data Capture and Extraction with C# Succinctly',
      page: this.pg }
  ];

  constructor(public navCtrl: NavController, public navParams: NavParams) {
  }

  ionViewDidLoad() {
    console.log('ionViewDidLoad BooksPage');
  }
}

```

Let's now work on updating the UI of the **Books** page. In order to do that, open the **books.html** file. Let's replace the existing markup with the following.

*Code Listing 4-o: The Updated books.html File*

```
<ion-header>
```

```

<ion-navbar>
  <ion-title>Books</ion-title>
</ion-navbar>
</ion-header>
<ion-content padding>
  <ion-list *ngFor="let book of booksData" no-padding>
    <ion-card>
      <ion-card-content>
        <ion-row center>
          <ion-col text-center>
            <h2><b>{{book.title}}</b></h2>
          </ion-col>
        </ion-row>
        <ion-row center>
          <ion-col text-center>
            <button ion-button [navPush]=book.page>
              Check book
            </button>
          </ion-col>
        </ion-row>
      </ion-card-content>
    </ion-card>
  </ion-list>
</ion-content>

```

As you can see, just like the **Paths** page, this page uses Ionic framework components such as **ion-card**, **ion-item**, **ion-card-content**, **ion-row**, and **ion-col**, among standard HTML markup.

To see how the **Books** page looks, let's execute the **ionic serve** command as we normally do.

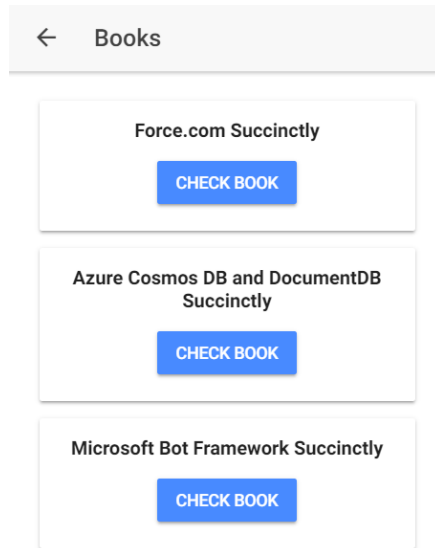


Figure 4-f: The Updated Books Page

Great—we’ve finished the **Books** page. Next on our list is the **Book Overview** page.

## The Book Overview page

We’ve reached an interesting milestone in building our app. Until now, all the pages we’ve created and updated have been pretty much self-contained, meaning that the objects and variables they use are found within the same Angular component that defines the page itself.

With the **Book Overview** page, this won’t be the case any longer. Because we have several book titles in our application’s library, it’s not practical to create an individual page for each one.

For a moment, just imagine we actually have all the *Succinctly* series titles in our app’s list. We would need to create more than 130 pages—this would not be practical or feasible.

In other words, the **Book Overview** page must be a generic page, to which we could pass as parameters the details of the book that we want the page to display.

Before we go and update the **Book Overview** page, we first need to talk a bit about navigation parameters within Ionic pages—also known as **navParams**.

## NavParams

Navigation parameters, just like the name implies, let us pass information from one page to another. This is incredibly useful, and defines the process of how we can send record-specific data to a page—which is exactly what we need in order to make the **Book Overview** page generic enough that it can be used to display data from any given book.

This also means we'll have to slightly modify the **Paths** and **Books** pages in order to pass the necessary navigation parameters for the books listed on both these pages.

But before we make any adjustments to the **Paths** and **Books** pages, let's first understand how navigation parameters actually work, and how they can be passed, by looking at some generic code samples.

The first way to specify navigation parameters is by adding a **navParams** directive, as can be seen in the following code.

*Code Listing 4-p: The navParams Directive*

```
<button ion-button [navPush]=book.page [navParams]=book.params>  
  Check book  
</button>
```

We are able to assign a **book.params** object to the **navParams** directive, thus passing the required navigation parameters to **book.page**.

The other way we can achieve this is by code, as shown in the following example.

*Code Listing 4-q: Passing navParams by Code*

```
pushPage() {  
  this.navCtrl.push(OtherPage, {name: 'Force.com Succinctly',  
    author: 'Ed Freitas', url: 'https://...'});  
}
```

Now that we know how to pass parameters to the page we are navigating to, how do we actually receive those parameters and do something with them?

The answer is very simple: basically, each Ionic-Angular page component constructor receives a **navParams** parameter, which contains all the navigation parameters passed on to the page.

All Ionic-Angular page components have this constructor. Here's an example.

*Code Listing 4-r: Page Component Constructor*

```
constructor(public navCtrl: NavController, public navParams: NavParams)  
{ }
```

Now that we know how to pass on parameters to pages and use them, we can appreciate how essential they are for us to be able to make pages generic.

Now, we're ready to get back to work and update the **Book Overview** page.



## Updating the Book Overview page

As its name implies, the **Book Overview** page will contain just a short overview of the book with some basic information, such as the author's name, date published, and the number of pages.

To understand the hierarchical structure of how the **Book Overview** (and also later, the **Book Detail**) page will display book information, let's look at the following diagram.



Figure 4-g: Hierarchical Book Information

With this in mind, let's update this page accordingly—we can do this by adding this basic information to the page's existing HTML markup, as follows.

Code Listing 4-s: Updated bookoverview.html File

```
<ion-header>
  <ion-navbar>
    <ion-title>Overview</ion-title>
  </ion-navbar>
</ion-header>
<ion-content padding>
  <ion-card>
    <ion-card-content>
      <ion-row center>
        <ion-col text-center>
          <h1>{{book.title}}</h1>
        </ion-col>
      </ion-row>
      <ion-row center>
        <ion-col text-center>
```

```

        <h2><b>Author:</b> {{book.overview.author}}</h2>
    </ion-col>
</ion-row>
<ion-row center>
    <ion-col text-center>
        <h3><b>Published:</b>
            {{book.overview.published}}</h3>
    </ion-col>
</ion-row>
<ion-row center>
    <ion-col text-center>
        <h3>{{book.overview.numpages}} pages</h3>
    </ion-col>
</ion-row>
<ion-row center>
    <ion-col text-center>
        <button ion-button
            [navPush]= book.overview.details.page
            [navParams]=book>
            Book details
        </button>
    </ion-col>
</ion-row>
</ion-card-content>
</ion-card>
</ion-content>

```

Based on this example, we can see that the page is using a **book** data object that contains the book's information. The properties specific to the book's overview, such as date published, number of pages, and author, are grouped under the **book.overview** property.

We can see that this page invokes the **Book Details** page—this is done by assigning **book.overview.details.page** to **navPush**.

Notice that the whole **book** object is passed to the **Book Details** page through **navParams**.

It's cool that we have our **Book Overview** page UI ready, but how does the **BookoverviewPage** class actually process the **navParams** passed from the **Books** or **Paths** pages?

This is possible because the **navParams.data** object—which contains the actual data passed through the navigation parameters—is assigned to a **book** object within the **constructor** of the **BookoverviewPage** class, as we can see in the following code.

Code Listing 4-t: Updated bookoverview.ts File

```
import { Component } from '@angular/core';
import { IonicPage, NavController, NavParams } from 'ionic-angular';

@IonicPage()
@Component({
  selector: 'page-bookoverview',
  templateUrl: 'bookoverview.html',
})
export class BookoverviewPage {
  public book;

  constructor(public navCtrl: NavController, public navParams: NavParams) {
    this.book = navParams.data;
  }

  ionViewDidLoad() {
    console.log('ionViewDidLoad BookoverviewPage');
  }
}
```

This explains how the **Book Overview** page is able to receive and use the **navParams** sent by the invoking page.

However, we still do not know how the **Books** or **Paths** page actually invoke the **Book Overview** page, and what exactly is passed as **navParams**—this is what we'll explore next.

## Adding navParams to the Books page

To be able to invoke the **Book Overview** page from the **Books** page, we need to be able to assign the **book** object to **navParams**.

So, in essence, our **Books** page HTML markup should now look as follows.

Code Listing 4-u: Updated books.html File

```
<ion-header>
  <ion-navbar>
    <ion-title>Books</ion-title>
  </ion-navbar>
</ion-header>
<ion-content padding>
```

```

<ion-list *ngFor="let book of booksData" no-padding>
  <ion-card>
    <ion-card-content>
      <ion-row center>
        <ion-col text-center>
          <h2><b>{{book.title}}</b></h2>
        </ion-col>
      </ion-row>
      <ion-row center>
        <ion-col text-center>
          <button ion-button
            [navPush]=book.page [navParams]=book>
            Overview
          </button>
        </ion-col>
      </ion-row>
    </ion-card-content>
  </ion-card>
</ion-list>
</ion-content>

```

As you can see, the only thing new is that on the **button**, the **book** object is being assigned to **navParams**—which is what the **BookoverviewPage** class later uses.

That's cool—however, let's not forget that the **Paths** page also invokes the **Book Overview** page, so in essence, both the **Books** and **Paths** pages should share the **book** object—as both pages need to use it. Any change to the **book** object would affect both pages.

To achieve this, we need to modify the **books.ts** file and make the **book** object accessible to the **Paths** page. But because we have more than one **book** object, what we really have is an array of **book** objects.

So let's define a **BooksData** array within **books.ts** that will contain the various **book** objects that our application will handle.

*Code Listing 4-v: The BookData Array*

```

export let BooksData = [
  { title: 'Force.com Succinctly', page: 'BookoverviewPage',
    overview: { author: 'Ed Freitas', numpages: 100,
    published: 'December 08, 2017',
    details: { page: 'BookdetailPage',
    pic:
    'https://cdn.syncfusion.com/content/images/downloads/ebook/ebook-
    cover/forcedotcomsuccinctly.jpg?v=09012018021233',

```

```

url: 'https://www.syncfusion.com/ebooks/forcedotcomsuccinctly' } } },

{ title: 'Azure Cosmos DB and DocumentDB Succinctly',
page: 'BookoverviewPage',
overview: { author: 'Ed Freitas', numpages: 103,
published: 'May 23, 2017',
details: { page: 'BookdetailPage',
pic:
'https://cdn.syncfusion.com/content/images/downloads/ebook/ebook-
cover/Azure_Cosmos_DB_and_DocumentDB_Succinctly.jpg?v=09012018021233',
url:
'https://www.syncfusion.com/ebooks/azure_cosmos_db_and_documentdb_succinctl
y' } } },

{ title: 'Microsoft Bot Framework Succinctly', page: 'BookoverviewPage',
overview: { author: 'Ed Freitas', numpages: 109,
published: 'May 02, 2017',
details: { page: 'BookdetailPage',
pic: 'https://cdn.syncfusion.com/content/images/downloads/ebook/ebook-
cover/Microsoft_Bot_Framework_Succinctly.jpg?v=09012018021233',
url:
'https://www.syncfusion.com/ebooks/azure_cosmos_db_and_documentdb_succinctl
y' } } },

{ title: 'Twilio with C# Succinctly', page: 'BookoverviewPage',
overview: { author: 'Ed Freitas', numpages: 107,
published: 'April 03, 2017',
details: { page: 'BookdetailPage',
pic: 'https://cdn.syncfusion.com/content/images/downloads/ebook/ebook-
cover/Twilio_with_C_sharp_Succinctly.jpg?v=09012018021233',
url: 'https://www.syncfusion.com/ebooks/twilio_with_c_sharp_succinctly' } }
},

{ title: 'Customer Success for C# Developers Succinctly', page:
'BookoverviewPage',
overview: { author: 'Ed Freitas', numpages: 97,
published: 'October 03, 2016',
details: { page: 'BookdetailPage',
pic: 'https://cdn.syncfusion.com/content/images/downloads/ebook/ebook-
cover/Customer_Success_Succicntly.jpg?v=09012018021233',
url:
'https://www.syncfusion.com/ebooks/customer_success_for_c_sharp_developers'
} } },

```

```
{ title: 'Data Capture and Extraction with C# Succinctly', page:
  'BookoverviewPage',
  overview: { author: 'Ed Freitas', numpages: 85,
  published: 'September 19, 2016',
  details: { page: 'BookdetailPage',
  pic: 'https://cdn.syncfusion.com/content/images/downloads/ebook/ebook-
  cover/Data_Capture_And_Extraction_Succinctly.jpg?v=09012018021233',
  url:
  'https://www.syncfusion.com/ebooks/data_capture_and_extraction_with_c_sharp
  _succinctly' } } }
];
```

Notice that each **book** object within the **BooksData** array contains general information, but also an **overview** object, and within that, a **details** object. So, essentially, every instance of **BooksData** array—each **book** object—contains the book's general information, the book's overview (which is used by the **BookoverviewPage** class), and the book's details (which will be used by the **BookdetailPage** class—we'll see this shortly).

This is how the modified books.ts file now looks.

*Code Listing 4-w: Updated books.ts File*

```
import { Component } from '@angular/core';
import { IonicPage, NavController, NavParams } from 'ionic-angular';

export let BooksData = [
  { title: 'Force.com Succinctly', page: 'BookoverviewPage',
  overview: { author: 'Ed Freitas', numpages: 100,
  published: 'December 08, 2017',
  details: { page: 'BookdetailPage',
  pic:
  'https://cdn.syncfusion.com/content/images/downloads/ebook/ebook-
  cover/forcedotcomsuccinctly.jpg?v=09012018021233',
  url: 'https://www.syncfusion.com/ebooks/forcedotcomsuccinctly' } } },

  { title: 'Azure Cosmos DB and DocumentDB Succinctly',
  page: 'BookoverviewPage',
  overview: { author: 'Ed Freitas', numpages: 103,
  published: 'May 23, 2017',
  details: { page: 'BookdetailPage',
  pic:
  'https://cdn.syncfusion.com/content/images/downloads/ebook/ebook-
  cover/Azure_Cosmos_DB_and_DocumentDB_Succinctly.jpg?v=09012018021233',
```

```

url:
'https://www.syncfusion.com/ebooks/azure_cosmos_db_and_documentdb_succinctl
y' } } },

{ title: 'Microsoft Bot Framework Succinctly', page: 'BookoverviewPage',
overview: { author: 'Ed Freitas', numpages: 109,
published: 'May 02, 2017',
details: { page: 'BookdetailPage',
pic: 'https://cdn.syncfusion.com/content/images/downloads/ebook/ebook-
cover/Microsoft_Bot_Framework_Succinctly.jpg?v=09012018021233',
url:
'https://www.syncfusion.com/ebooks/azure_cosmos_db_and_documentdb_succinctl
y' } } },

{ title: 'Twilio with C# Succinctly', page: 'BookoverviewPage',
overview: { author: 'Ed Freitas', numpages: 107,
published: 'April 03, 2017',
details: { page: 'BookdetailPage',
pic: 'https://cdn.syncfusion.com/content/images/downloads/ebook/ebook-
cover/Twilio_with_C_sharp_Succinctly.jpg?v=09012018021233',
url: 'https://www.syncfusion.com/ebooks/twilio_with_c_sharp_succinctly' } }
},

{ title: 'Customer Success for C# Developers Succinctly', page:
'BookoverviewPage',
overview: { author: 'Ed Freitas', numpages: 97,
published: 'October 03, 2016',
details: { page: 'BookdetailPage',
pic: 'https://cdn.syncfusion.com/content/images/downloads/ebook/ebook-
cover/Customer_Success_Succinctly.jpg?v=09012018021233',
url:
'https://www.syncfusion.com/ebooks/customer_success_for_c_sharp_developers'
} } },

{ title: 'Data Capture and Extraction with C# Succinctly', page:
'BookoverviewPage',
overview: { author: 'Ed Freitas', numpages: 85,
published: 'September 19, 2016',
details: { page: 'BookdetailPage',
pic: 'https://cdn.syncfusion.com/content/images/downloads/ebook/ebook-
cover/Data_Capture_And_Extraction_Succinctly.jpg?v=09012018021233',
url:
'https://www.syncfusion.com/ebooks/data_capture_and_extraction_with_c_sharp
_succinctly' } } }

```

```

];

@IonicPage()
@Component({
  selector: 'page-books',
  templateUrl: 'books.html',
})
export class BooksPage {
  booksData = BooksData;

  constructor(public navCtrl: NavController, public navParams: NavParams)
  { }

  ionViewDidLoad() {
    console.log('ionViewDidLoad BooksPage');
  }
}

```

Notice how the **BooksData** array is assigned to the **booksData** object, which is used in **books.html** within the **ngFor** directive on the **ion-list** component, in order to iterate through each of the books.

## Adding navParams to the Paths page

To be able to invoke the **Books Overview** page from the **Paths** page, we need to be able to assign the **book** object to **navParams**.

The **Paths** page HTML markup should now look as follows.

*Code Listing 4-x: Updated paths.html File*

```

<ion-header>
  <ion-navbar>
    <ion-title>Paths</ion-title>
  </ion-navbar>
</ion-header>
<ion-content padding>
  <ion-list *ngFor="let path of pathData" no-padding>
    <ion-card>
      <ion-item>
        <ion-row center>
          <ion-col text-center>
            <h1><b>{{path.name}}</b></h1>

```



```

        </ion-col>
      </ion-row>
    </ion-item>
    <ion-list *ngFor="let book of path.books" no-padding>
      <ion-card-content>
        <ion-row center>
          <ion-col text-center>
            <h2>{{book.title}}</h2>
          </ion-col>
        </ion-row>
        <ion-row center>
          <ion-col text-center>
            <button ion-button [navPush]=book.page
              [navParams]=book>
              Check book
            </button>
          </ion-col>
        </ion-row>
      </ion-card-content>
    </ion-list>
  </ion-card>
</ion-list>
</ion-content>

```

Now that we've assigned the **book** object to **navParams**, let's refactor a bit the code we previously wrote for **paths.ts**.

We need to do this in order to make use of the **BooksData** array that we created within **books.ts**, so we can use it inside **paths.ts**. We can do this by also importing **BooksData** within our **BooksPage** import clause. The following is the refactored **paths.ts** file.

*Code Listing 4-y: Updated paths.ts File*

```

import { Component } from '@angular/core';
import { IonicPage, NavController, NavParams } from 'ionic-angular';
import { BooksPage, BooksData } from '../pages/books/books';

@IonicPage()
@Component({
  selector: 'page-paths',
  templateUrl: 'paths.html',
})
export class PathsPage {
  pg = 'BookoverviewPage';
}

```

```

pathData = [
    { name: 'Database',
      books: [
        BooksData[0],
        BooksData[1]
      ]
    },
    { name: 'Web Development',
      books: [
        BooksData[0]
      ]
    },
    { name: 'Cloud',
      books: [
        BooksData[1]
      ]
    },
    { name: 'Microsoft & .NET',
      books: [
        BooksData[2],
        BooksData[3],
        BooksData[4],
        BooksData[5]
      ]
    },
    { name: 'Mobile Development',
      books: [
        BooksData[3]
      ]
    },
    { name: 'Miscellaneous',
      books: [
        BooksData[5]
      ]
    }
];

constructor(public navCtrl: NavController, public navParams: NavParams)
{ }

ionViewDidLoad() {
    console.log('ionViewDidLoad PathsPage');
}
}

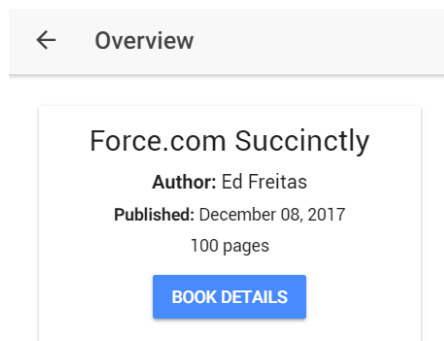
```

Notice that I've refactored the **books** array objects within **pathData** by using the **BooksData** object imported from the **Books** page. This way, we don't need to repeat any book data.

In a real-life scenario, you'll probably fetch a similar data structure like **BooksData** from a web server using a RESTful API, with an AJAX request through an Angular service. However, the same focus on modularity and reusability should be kept.

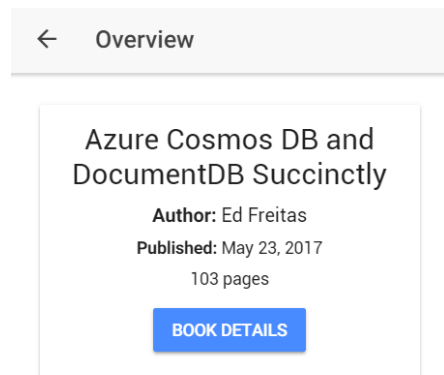
We have now updated all the existing pages that invoke the **Book Overview** page. The only remaining bit of our application is to update the **Book Detail** page, which we'll do shortly.

Before we do that, let's run the **ionic serve** command once again to check how the **Book Overview** page looks when accessed from the **Paths** page.



*Figure 4-h: The Book Overview Page Invoked from the Paths Page*

Now, let's do the same thing, but by navigating to the **Book Overview** page from the **Books** page.



*Figure 4-i: The Book Overview Page Invoked from the Books Page*

We can see that we were able to navigate to the **Book Overview** page for two different books: one through the **Paths** page, and the other through the **Books** page.

The only thing we need to do to finalize our application is update the **Book Detail** page, which is what we'll do next.

## Updating the Book Detail page

We're on the last stretch of this app. The **Book Detail** page, as its name clearly implies, is responsible for displaying the properties of the `details` object for each book.

In order to do that, let's modify our existing `bookdetail.html` file and add the following markup.

*Code Listing 4-z: Updated bookdetail.html File*

```
<ion-header>
  <ion-navbar>
    <ion-title>Details</ion-title>
  </ion-navbar>
</ion-header>
<ion-content padding>
  <ion-card>
    <ion-card-content>
      
      <ion-row center>
        <ion-col text-center>
          <button ion-button>
            Get book (For future use)</button>
          </ion-col>
        </ion-row>
      </ion-card-content>
    </ion-card>
  </ion-content>
```

As you've probably noticed, we are only displaying the `pic` property. This is because I'd like to leave you a challenge to figure out on your own.

The challenge is to use the `templateUrl` property and add it to the button, so that when clicking on it in a browser window, it can be opened and you can grab the book directly from the Syncfusion website.

I'll give you a hint—you'll be able to do this by launching in App Browser—this is possible by using the InAppBrowser Cordova plugin. More details can be found [here](#), so feel free to have a look.

With the UI of the **Book Detail** page done, let's now move on and focus on the `bookdetail.ts` file, and update it accordingly. We can see the updated code in Code Listing 4-aa.

*Code Listing 4-aa: Updated bookdetail.ts File*

```
import { Component } from '@angular/core';
import { IonicPage, NavController, NavParams } from 'ionic-angular';
```

```

@IonicPage()
@Component({
  selector: 'page-bookdetail',
  templateUrl: 'bookdetail.html',
})
export class BookdetailPage {
  public book;

  constructor(public navCtrl: NavController, public navParams: NavParams) {
    this.book = navParams.data;
  }

  ionViewDidLoad() {
    console.log('ionViewDidLoad BookdetailPage');
  }
}

```

All we are really doing here is getting the **navParams.data** object (the actual navigation parameters received from the **Book Overview** page) and assigning the parameters to the **book** object, which is what the bookdetail.html uses.

Let's now execute the **ionic serve** command once again in order to view the changes we've done.

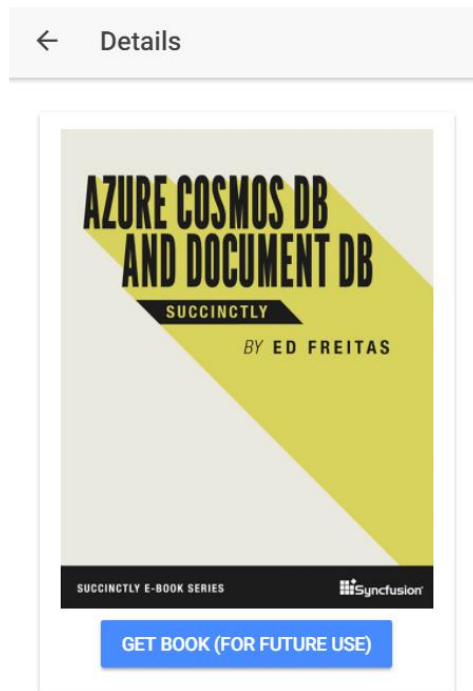


Figure 4-j: The Book Detail Page

Awesome—we now have a fully working app. We’ve gone through all the steps in order to create a small, but fully working Ionic application—congrats!

## Full source code

You can download the full source code of this app [here](#). Please note that this includes the complete project, which can be opened in your development environment of choice—in my case, I used Visual Studio Code.

Also note that this project doesn’t include the `\node_modules` folder, so in order to install the required Node.js modules, you need to go into the project folder and run the **`npm install -g ionic cordova`** command.

## Summary

This has been a long but interesting chapter, as we’ve seen how to navigate through the various app pages and give the application its character and functionality.

Given the breadth of the Ionic framework, it’s impossible to cover it all in one short book. However, with the step-by-step approach we’ve taken to build this app, we’ve explored most of the essentials you’ll need in order to get off the ground with this framework and create your own app.

In the next and final chapter, we’ll quickly review various Ionic resources that should help you expand your understanding of this framework, so you can keep exploring it on your own.

# Chapter 5 Further Resources

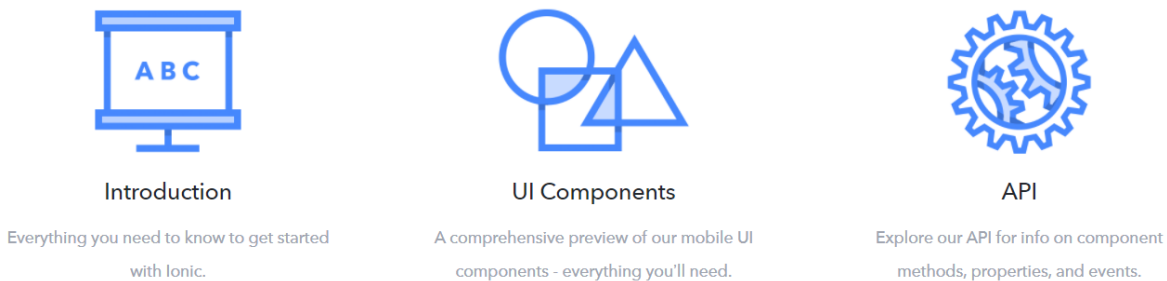
## Quick intro

There's so much more to the Ionic framework that I'd probably need at least two or three more books to cover it all, and who knows—I might write a follow-up on this topic in the future. But for now, I think we've managed to achieve the goal of covering how to get started and build something without adding unnecessary complexity.

In any case, I'd like to quickly go over some extra Ionic framework resources and details that might come in handy while you keep exploring this fascinating ecosystem on your own.

## Exploring the documentation

Needless to say, the Ionic [documentation](#) is the definitive resource from all points of view. It is organized into various sections that make it incredibly easy for anyone to find relevant information.



*Figure 5-a: Screenshot of the Ionic Documentation Site*

Here are the main sections the documentation covers:

- Introduction: Getting started with the framework.
- UI Components: Reference guide for all the available components.
- API Reference: Up-to-date details of all the methods, properties, and events.
- Ionic Native: How to integrate native functionalities and use plugins.
- Theming: How to change the look and feel of your app.
- Ionicons (Icons): Extensive information on all Ionic custom-designed icon resources.
- CLI: Complete information on how to use the Command Line Interface.

Beyond these topics, the documentation site also provides an extensive FAQ and hosts a forum where users ask all sort of questions of the Ionic technical team and get feedback on their queries.

Overall, the documentation is an invaluable resource.

## Native plugins

An Ionic app running on a browser will not have access to the hardware on the device. A great thing about Ionic is that it has plugins to get around these limitations.

These plugins are usually referred to as native plugins because they give you access to the hardware present on the device.



*Figure 5-b: A “Plugin” Icon*

As time has passed, these plugins have expanded beyond offering only access to the hardware, by also providing many different features that you would expect from a mobile app.

Many of these plugins can only be tested directly on a device, and may fail while being tested on a browser. The reason is that many of these plugins might be looking for a hardware functionality that only exists on the device itself.

One of the most well-known or often-used plugins is the one that provides access to the device’s camera. Although we won’t be adding any further code to our application to make use of the camera, it’s useful to know which commands you may use in case you feel like playing around with it.

*Code Listing 5-a: Commands to Install the Camera Plugin*

```
ionic cordova plugin add cordova-plugin-camera
npm install --save @ionic-native/camera
```

The execution of these commands lets us install the necessary Cordova plugin that gives us access to our device’s camera, allowing us to capture still images and video.

Once installed, you can use the following code in order to import the plugin and inject the **Camera** object—this is done within a TypeScript file.

*Code Listing 5-b: Importing the Camera Plugin*

```
import { Camera, CameraOptions } from '@ionic-native/camera';

export class SomeClass {
  constructor(private camera: Camera)
  { }
}
```



```
}
```

The example that follows illustrates how we can supply the camera options to the **getPicture** method.

*Code Listing 5-c: Supplying the Camera Options to the getPicture Method*

```
const options = CameraOptions = {
  destinationType: this.camera.DestinationType.DATA_URL,
  mediaType: this.camera.MediaType.PICTURE
};

this.camera.getPicture(options).then((imageData) => {
  let base64Image = 'data:image/jpeg;base64,' + imageData
}, (err) => {
});
```

The camera options allow us to control such things as the image or the video, instead.

There are many other plugins as well—a common one is the push plugin. The use of plugins within Ionic is done with the Ionic Native library.

Ionic Native is a TypeScript wrapper for Cordova/PhoneGap plugins that make it easy to add any native functionality you need to your Ionic mobile app.

I highly encourage you to investigate all the possibilities of using native plugins by checking out the official Ionic Native [documentation](#).

## Search bar

Something we can potentially add to our application is a search bar, which has become an expected component in most apps these days.

If we look at the app's main screen, it's not easy to find any of the books that are listed within the library quickly.

In order to find a book, we have to either browse through the **Paths** or **Books** pages—which is great, but if we had to browse through 100 or more books, it would be a bit tedious.

To make it easier to search for a specific book, let's add the search bar to the **Library** page of the app—this will allow us to navigate to the **Book Overview** page of that book directly.

Here's how an Ionic search bar looks within the markup code.

Code Listing 5-d: An Ionic Search Bar

```
<ion-searchbar placeholder="Search titles, press enter"
  [(ngModel)]="queryText" (search)="updateBooks()">
</ion-searchbar>
```

As we can see, the search bar has some attributes: the **placeholder**, the **ngModel** query text, and the **search**, which is the method that will get executed when Enter is pressed.

Before we actually add any functionality, let's see how it looks on the screen. We can do this by adding that code to the **library.html** file as follows, and then running the **ionic serve** command.

Code Listing 5-e: The Updated library.html File

```
<ion-header>
  <ion-navbar>
    <button ion-button menuToggle>
      <ion-icon name="menu"></ion-icon>
    </button>
    <ion-title>Succinctly Library</ion-title>
  </ion-navbar>
</ion-header>
<ion-content padding>
  <ion-searchbar placeholder="Search titles, press enter "
    [(ngModel)]="queryText" (search)="updateBooks()">
  </ion-searchbar>
  <ion-list *ngFor="let card of cardData" no-padding>
    <ion-card>
      <ion-item>
        <h2>{{card.header}}</h2>
        <p>{{card.short}}</p>
      </ion-item>
      
      <ion-card-content>
        <ion-card-title>
          {{card.title}}
        </ion-card-title>
        <p>{{card.description}}</p>
      </ion-card-content>
      <ion-row center>
        <ion-col text-center>
          <button ion-button [navPush]=card.pushPage>
            {{card.buttontxt}}
          </button>
        </ion-col>
      </ion-row>
    </ion-card>
  </ion-list>
</ion-content>
```

```

        </ion-col>
      </ion-row>
    </ion-card>
  </ion-list>
</ion-content>

```

The following screenshot shows how the **Library** page now looks.

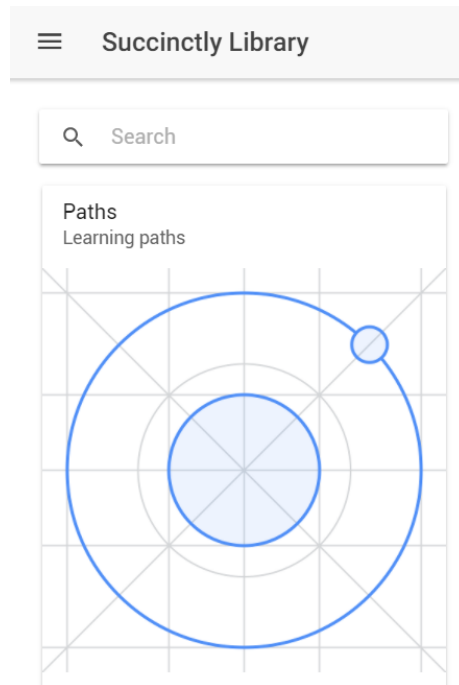


Figure 5-c: The Library Page with a Search Bar

We now have a nice-looking search bar, but it doesn't do anything yet. Next, we'll add the actual search functionality to it.

## Adding the search functionality

With the `library.html` file updated, the next thing we have to do is add the logic for our search functionality.

This means we'll have to update the **library.ts** file first, and then also update **books.ts**, as well. Let's first define the logic behind the **updateBooks** method, as follows.

Code Listing 5-f: The `updateBooks` Method

```
existsIn(str)
```

```

{
  let res = false;
  for (var i = 0; i < this.booksData.length; i++) {
    if (this.booksData[i].title.toLowerCase().
      indexOf(str.toLowerCase()) >= 0) {
      res = true;
      break;
    }
  }
  return res;
}

itemExists(haystack, needle)
{
  for (var i = 0; i < haystack.length; i++) {
    if (haystack[i].title.toLowerCase().
      indexOf(needle.toLowerCase()) >= 0) {
      if (!this.existsIn(haystack[i].title)) {
        this.booksData.push(haystack[i]);
      }
    }
  }
}

updateBooks()
{
  this.booksData = [];
  if (this.queryText !== undefined) {
    let qry = this.queryText.toLowerCase();
    this.itemExists(BooksData, qry);
    if (this.booksData.length > 0) {
      this.navCtrl.push(BooksPage, this.booksData);
    }
  }
}

```

The **updateBooks** method basically checks if **this.queryText** exists as one of the book titles. This is done by the **itemExists** method, which essentially loops through all the elements of the **BooksData** array and verifies which of those titles contains **this.queryText**.

If the **itemExists** method is able to find a match, then it verifies that the matching element of **BooksData** doesn't exist yet within **this.booksData**—so that no repeated elements are added to **this.booksData**.

The **this.booksData** array is what will be passed on as **navParams** to the **Books** page when Enter is pressed.

As you can see, all this logic is pretty straightforward and uncomplicated. Now that this has been explained, let's have a look at the updated library.ts file.

*Code Listing 5-g: The Updated library.ts File*

```
import { Component } from '@angular/core';
import { IonicPage, NavController, NavParams } from 'ionic-angular';
import { PathsPage } from '../pages/paths/paths';
import { BooksPage, BooksData } from '../pages/books/books';

@IonicPage()
@Component({
  selector: 'page-library',
  templateUrl: 'library.html',
})
export class LibraryPage {
  queryText: string;
  booksData = [];

  cardData = [
    { img: '../assets/imgs/logo.png',
      header: 'Paths',
      short: 'Learning paths',
      title: '',
      description: 'Paths are books organized by related topics',
      buttontxt: 'Explore Paths',
      pushPage: 'PathsPage'
    },
    { img: '../assets/imgs/azure.jpg',
      header: 'Books',
      short: 'Books to read',
      title: '',
      description: 'Over 130 books on technologies that matter',
      buttontxt: 'Browse Books',
      pushPage: 'BooksPage'
    }
  ];

  existsIn(str) {
    let res = false;
    for (var i = 0; i < this.booksData.length; i++) {
      if (this.booksData[i].title.toLowerCase().
```

```

        indexOf(str.toLowerCase()) >= 0) {
            res = true;
            break;
        }
    }
    return res;
}

itemExists(haystack, needle) {
    for (var i = 0; i < haystack.length; i++) {
        if (haystack[i].title.toLowerCase().
            indexOf(needle.toLowerCase()) >= 0) {
            if (!this.existsIn(haystack[i].title)) {
                this.booksData.push(haystack[i]);
            }
        }
    }
}

updateBooks() {
    this.booksData = [];
    if (this.queryText !== undefined) {
        let qry = this.queryText.toLowerCase();
        this.itemExists(BooksData, qry);
        if (this.booksData.length > 0) {
            this.navCtrl.push(BooksPage, this.booksData);
        }
    }
}

constructor(public navCtrl: NavController, public navParams: NavParams)
{ }

ionViewDidLoad() {
    console.log('ionViewDidLoad LibraryPage');
}
}

```

As we can see, besides the **updateBooks** method-related modifications, the only other changes within the **library.ts** file are that we imported the original **BooksData** array from the **Books** page, declared **queryText**, and initialized **this.booksData**.

In order for this to actually work, we need to make the **Books** page aware that it may receive data from the **Library** page through the search bar.

So on the **constructor** of the **books.ts** file, we need to add some code in order to verify whether the **navParams.data** is actually not empty.

By default, the **Books** page displays all the books in the library, so when it is invoked from the **Library** page with the **BROWSE BOOKS** button, all the elements of the **BooksData** array are displayed. This means that, in this case, **navParams.data** is empty.

However, when the **Books** page is invoked from the search bar on the **Library** page, only a subset of the **BooksData** array is passed on to the **Books** page as **navParams**—which means that **navParams.data** is not empty.

Therefore, the **constructor** needs to be modified to handle this new scenario, as follows.

*Code Listing 5-h: The Updated Constructor of books.ts*

```
constructor(public navCtrl: NavController, public navParams: NavParams)
{
    this.booksData = !this.isEmpty(navParams.data) ?
        navParams.data : BooksData;
}
```

To check whether **navParams.data** is empty, we'll need to invoke another method, called **isEmpty**, which is coded as follows.

*Code Listing 5-i: The isEmpty Method*

```
isEmpty(obj)
{
    for (var key in obj) {
        if (obj.hasOwnProperty(key))
            return false;
    }
    return true;
}
```

This method simply loops through each of the key-value pairs of the object and checks whether a **key** has a corresponding value—**hasOwnProperty**.

With this additional logic, the updated **books.ts** file now looks as follows.

*Code Listing 5-j: The Updated books.ts File*

```
import { Component } from '@angular/core';
import { IonicPage, NavController, NavParams } from 'ionic-angular';

export let BooksData = [
    { title: 'Force.com Succinctly', page: 'BookoverviewPage',
```

```

overview: { author: 'Ed Freitas', numpages: 100,
published: 'December 08, 2017',
details: { page: 'BookdetailPage',
pic: 'https://cdn.syncfusion.com/content/images/downloads/ebook/ebook-
cover/forcedotcomsuccinctly.jpg?v=09012018021233',
url: 'https://www.syncfusion.com/ebooks/forcedotcomsuccinctly' } } },

{ title: 'Azure Cosmos DB and DocumentDB Succinctly', page:
'BookoverviewPage',
overview: { author: 'Ed Freitas', numpages: 103,
published: 'May 23, 2017',
details: { page: 'BookdetailPage',
pic: 'https://cdn.syncfusion.com/content/images/downloads/ebook/ebook-
cover/Azure_Cosmos_DB_and_DocumentDB_Succinctly.jpg?v=09012018021233',
url:
'https://www.syncfusion.com/ebooks/azure_cosmos_db_and_documentdb_succinctl
y' } } },

{ title: 'Microsoft Bot Framework Succinctly', page: 'BookoverviewPage',
overview: { author: 'Ed Freitas', numpages: 109,
published: 'May 02, 2017',
details: { page: 'BookdetailPage',
pic: 'https://cdn.syncfusion.com/content/images/downloads/ebook/ebook-
cover/Microsoft_Bot_Framework_Succinctly.jpg?v=09012018021233',
url:
'https://www.syncfusion.com/ebooks/azure_cosmos_db_and_documentdb_succinctl
y' } } },

{ title: 'Twilio with C# Succinctly', page: 'BookoverviewPage',
overview: { author: 'Ed Freitas', numpages: 107,
published: 'April 03, 2017',
details: { page: 'BookdetailPage',
pic: 'https://cdn.syncfusion.com/content/images/downloads/ebook/ebook-
cover/Twilio_with_C_sharp_Succinctly.jpg?v=09012018021233',
url: 'https://www.syncfusion.com/ebooks/twilio_with_c_sharp_succinctly' } }
},

{ title: 'Customer Success for C# Developers Succinctly', page:
'BookoverviewPage',
overview: { author: 'Ed Freitas', numpages: 97,
published: 'October 03, 2016',
details: { page: 'BookdetailPage',
pic: 'https://cdn.syncfusion.com/content/images/downloads/ebook/ebook-
cover/Customer_Success_Succinctly.jpg?v=09012018021233',

```



```

url:
'https://www.syncfusion.com/ebooks/customer_success_for_c_sharp_developers'
} } },

{ title: 'Data Capture and Extraction with C# Succinctly', page:
'BookoverviewPage',
overview: { author: 'Ed Freitas', numpages: 85,
published: 'September 19, 2016',
details: { page: 'BookdetailPage',
pic: 'https://cdn.syncfusion.com/content/images/downloads/ebook/ebook-
cover/Data_Capture_And_Extraction_Succinctly.jpg?v=09012018021233',
url:
'https://www.syncfusion.com/ebooks/data_capture_and_extraction_with_c_sharp
_succinctly' } } }
];

@IonicPage()
@Component({
  selector: 'page-books',
  templateUrl: 'books.html',
})
export class BooksPage {
  booksData = BooksData;

  isEmpty(obj) {
    for (var key in obj) {
      if (obj.hasOwnProperty(key))
        return false;
    }
    return true;
  }

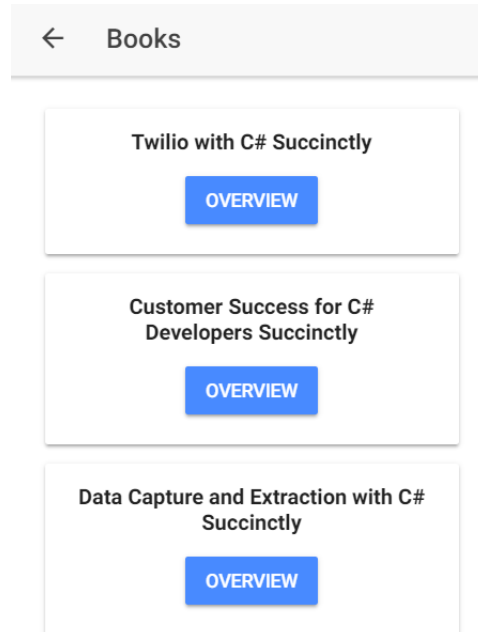
  constructor(public navCtrl: NavController, public navParams: NavParams)
  {
    this.booksData = !this.isEmpty(navParams.data) ?
      navParams.data : BooksData;
  }

  ionViewDidLoad() {
    console.log('ionViewDidLoad BooksPage');
  }
}

```

We are done making all the changes needed to have our search bar ready. Let's now run the **ionic serve** command and check out our search bar's functionality.

In order to test things properly, I'll type in **C#** and press **Enter**. I should expect to get three results—let's see if that's the case.



*Figure 5-d: The Search Bar Results*

Great—it works as expected. By simply adding an extra bit of functionality and reusing the code we already had, we were able to add a really cool and useful feature to our application.

## Full, updated source code

The full, updated source code of our application, including the most recent changes with the search bar, can be downloaded [here](#).

## Other topics to explore

Although we've covered quite a lot of ground, there's still a lot to explore and learn about the amazing Ionic ecosystem.

The following is a list of items I suggest you look at and investigate on your own time. Although the list is far from being conclusive, it includes some nice aspects that, in my opinion, are worth delving into:

- Geolocation with Ionic Native.
- Vibration and motion detection with Ionic Native.

- Barcode scanning using Ionic Native.
- Notifications with Ionic Native.
- Integrating SQLite into your app.
- Deployment and testing on native platforms: iOS and Android.

## Summary

Ionic is a mature, solid, easy-to-learn, and state-of-the-art development framework for mobile and Progressive Web Apps. What can be done with this framework with a few lines of code is remarkable.

The goal of this book was to walk you through a simple but interesting example on how to build a useful application with Ionic, without requiring any previous knowledge of Ionic 1 or Angular. I think we achieved this objective and had some fun along the way.

This book is by no means an Ionic “bible,” as the length, depth, and extension of the framework would require many more pages than what we could cover at this point in time—but we did manage to introduce and show what you can do with this framework with a few simple steps.

I really hope you enjoyed reading this book as much as I enjoyed writing it, and I hope this serves as a source of inspiration for you to keep exploring this fascinating technology.

Thank you, and until next time, all the best.