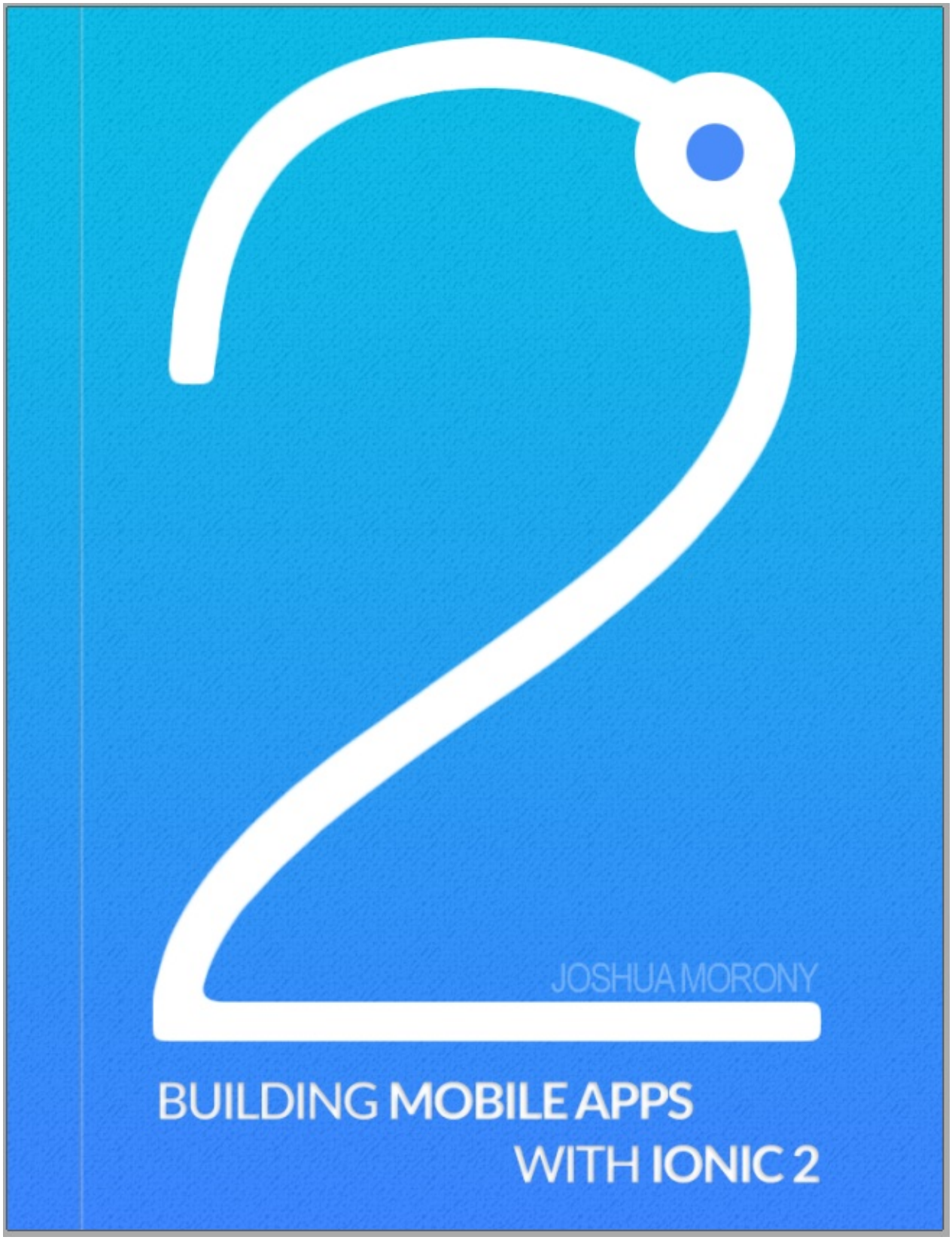
目錄

使用Ionic 2制作移动APP

本书可以叫做 Ionic 2从入门到精通。
除了介绍Ionic 2，TypeScript基本知识之外，
手把手的教会大家制作了5个应用，
以及打包应用，发布应用，更新应用。
但是有可能的话请大家支持正版，
大家都不容易。



工作日志：

- 2017-1-3 开始建立目录
- 2017-5-5 开始翻译工作
- 2017-5-11 14:43 已经翻译到第一章第9节
- 2017-5-22 自嗨中，估计没多少人看，哇哈哈

- 2017-5-25 最近感觉翻译得好生硬，好像我要稍微浪一点
- 2017-6-1 初步完成，懒得校对了

第一章 介绍

欢迎！

欢迎学习使用**Ionic 2** 制作移动应用！本书会教会你关于Ionic 2的一切知识，从基础入门到制作应用发布到App商店。
阅读本书的人们也许角度会有所不同，有些人已经熟悉Ionic 1，有些人已经在开始体验Ionic 2，有些人可能都不知道。 无论你在哪一个水准，可能都不重要。因为本书所有课程都解释得明白彻底，没有对Ionic的任何猜测。
同时本书不介绍HTML,CSS和JavaScript。在开始学习本书之前，你需要对这些知识有基础的了解。如果想要重新温习一下这些知识的话，建议你看看以下网站：

- 学习HTML & CSS
- 学习JavaScript

本书有很多不同的部分，但是有三个不同的区域。我们有基础知识入手，然后进入到应用制作，最后是打包和提交应用。
本书的所有应用范例都是完全独立的。虽然随着进度推进，每个应用的复杂度一点一点的在增加，我都是基于在你没有学习之前的范例的假设之上解释每个范例的， 所以每个范例中会有一些重复的信息存在。
注意：如果你购买本书的时候附送了视频教程，我建议你先看完视频教程再阅读本书。这不是强制要求，但是他只是一个基本介绍，而不是作为一个普通进度去学习。

更新与勘误

Ionic 2目前还在开发中，意思是他还在变。他目前是相对稳定的，所以阅读本书的大部分内容是不会变动的，但是照目前来看直到发布版放出之前会有变动的。（译者已经运行了本书的代码，译者的Ionic 2版本是3.1.1，可用的源代码已经在本译文的github同repo下） 我会经常根据框架的变动来更新本书，这些更新都是免费的。我更新本书的时候你应该可以收到新的下载链接的邮件通知。
我会保持关注更新和保证事务正常，但是，这是个大工程，如果你找到任何错问，请给我发邮件，我会尽快更新。

本书使用的一些约定

本书使用的布局其实不要解释太多，尽管如此，你还是要看一下：

> 这样的区块

意思是你要去执行的行为。例如，这些区块文本会告诉你去创建一个文件或者做一些代码变更。这些区块在范例里面很常见。这个语法很有用，因为他帮我向你区别展示在应用中需要变动的代码。
注意：你会遇到这样的区块。他包含了一些你当前做的事情的相关信息。
重要：你也会遇到一些这样的东西。这些都是一些需要特别注意的“陷阱”。
好了，讲的太多！开始动手吧！祝你好运！

更新日志

Version 12（本版） - 更新RC.3

- 少许bug fix和优化
- Giflist视频播放bug问题修复

..（都是本书内容更新日志，先放着）

新概念

Ionic 1是建立在Angular 1上的，Angular 1是用来制作复杂和伸展性的JavaScript应用的。Ionic基于Angular做的事情是提供了一系列的功能来使制作移动应用更简单。 然后，随着新一代的Angular 2的到来，带来了一系列的改动和改进。 Ionic如果想要使用Angular 2的话，那么也需要有所改变，然后这就是Ionic 2了。 简单来讲，得益于新的web标准，使用Ionic 2 和Angular 2我们可以让app在移动终端表现得更好，伸缩性，重用性，模组功能等等。
由于引入了Angular2，开发应用的方式来很大的改变。有大量的概念变更，以及有些东西的少量变更，例如模板语法。
在Ionic 2中，模板看起来是这样的：

```
<ion-menu [content]="content">
    <ion-toolbar>
        <ion-title>Pages</ion-title>
    </ion-toolbar>
    <ion-content>
        <ion-list>
            <button ion-item *ngFor="let p of pages" (click)="openPage(p)"></button>
        </ion-list>
    </ion-content>
</ion-menu>
<ion-nav id="nav" [root]="rootPage" #content></
```

和Ionic 1差别很大，你的JavaScript应该是这样子的：

```
import { Component } from '@angular/core';
import { Platform } from 'ionic-angular';
import { HomePage } from './pages/home/home';

@Component({
    template: `<ion-nav [root]="rootPage"></ion-nav>`
})

export class MyApp {
    rootPage: any = HomePage;
    constructor(platform: Platform) {
        platform.ready().then(() => {

        });
    }
}
```

跟Ionic 1比起来差别相当的大。但是如果你熟悉EXMAScript 6或者TypeScript的话，这些变动对你来说应该不是什么难事， 但是，如果这对你来说是全新的概念的话（大部分人都是这样）这个转变可能会有一丢丢难度。 为帮助你的思想负担转变减轻负担，Ionic 2 aplha版本刚出来的时候，ES6和TypeScript对我来说是全新的概念， 但是在短暂的一个时期之后，我就如鱼得水了。现在我对新的语法和结构的感觉比Ionic 1更舒服。
本课内我们将广泛的设计到Ionic 2和Angular 2的新概念和语法。目的是给你一个初步的认识，我们后续会详细介绍。

ECMAScript 6 （ES6)

在了解ECMAScript 6之前，我们需要了解一下ECMAScript是什么。追溯历史可能有点太复杂，但是核心是：**EXMAScript**是一个标准， **JavaScript**实现了这个标准。ECMAScript定义标准，浏览器执行这个标准。同样的，HTML标准（最新的HTML5）也是这个组织定义的，然后由浏览器提供商实现。 不同浏览器以不同的方法去实现这些标准，不同的功能有大量的不同的支持，这就是为什么有些东西在不同的浏览器里面表现不同。
HTML5规格是一个游戏规则改变者，某种程度上ECMAScript 6也是。 他将会为你的JavaScript代码带来非常巨大的改变，总的来讲，他使JavaScript编程更成熟的一门编程语言，可以更简单的hold大型的复杂的应用（这是JavaScirpt之前想都不敢想的）。
此刻我们不会继续深入ES6，因为这不是本书的主旨，但是我会给你展示几个示例，以便你感受一下。ES6引入到JavaScript的新功能包括：

类 Class

```
class Shape {
    constructor (id, x, y) {
        this.id = id
        this.move(x, y)
    }
    move (x, y) {
        this.x = x
        this.y = y
    }
}
```

这是个非常大的变动，如果你有过传统编程语言例如Java，C#的使用经验的话你应该非常熟悉。长久以来，许多人利用JavaScirpt的function来实现仿class结构，从来没有一个办法在JavaScript中使用真正的类。现在，他来了。如果你不知道什么是类，不要紧，后面有一整节课来讲解。

模组/模块 **Module**

```
// lib/math.js
export function sum (x, y) { return x + y }
export var pi = 3.141593

// someApp.js
import * as math from "lib/math"
console.log("2PI = " + math.sum(math.pi, math.pi))

// otherApp.js
import { sum, pi } from "lib/math"
console.log("2PI = " + sum(pi, pi))
```

模块功能允许你将代码代码，然后在任何其他地导入使用，这个功能在Ionic中会大量用到。我们后续会继续深入，实际上我们在应用中创建任何组件我们都会导出然后在任何地方导入。

Promise

Promise是之前通过服务（serivce）实现过的东西，例如之前的ngCordova，但是他们现在是原生支持的，意思就是你可以这样去写：

```
doSomething().then((response) => {
    console.log(response);
});
```

块语法

目前来讲，如果你JavaScript中定一个变量，他就在定义他的函数内随处可用。ES6新的块语法范围功能允许你通过*let*关键字来定义一些块内变量，如下：

```
for (let i = 0; i < a.length; i++) {
    let x = a[i];
}
```

如果你想在循环体外访问x变量的话，那么你就会得到undefined。

大箭头函数

> 译注：=> 相对于 -> c++里用到的

我最喜欢的功能之一是大箭头函数，你可以这么去浪：

```
someFunction((response) => {
    console.log(response);
});
```

而不是之前这样：

```
someFunction(function(response){
    console.log(response);
});
```

初次看到，你可能会觉得不咋滴，但是他可以让你保持函数中父类的范围内。 上面的范例中，当我们访问**this**关键字的时候，你就会访问到本体，但是在下面的函数中，你想要访问到本体的话需要这么做：

```
var me = this;
someFunction(function(response){
    console.log(me.someVariable);
});
```

得益于新语法，我们就不需要创建一个**this**的静态引用了，我们可以直接使用**this**。
想要了解ES6的新功能，请查看这里：http://es6-features.org/

TypeScript

另一个需要浏览一下的改变是TypeScript，我们在Ionic 2中需要用到。需要重点指出的是尽管Ionic 2用来TypeScript，你不一定需要用他来制作Ionic 2应用，你可以可以用纯ES6。话虽如此，TypeScript提供了额外的功能（特别是依赖注入）使事情简单化，很快他将会成为Ionic 2

的标配，所以说不用他说没有道理的。
本书将使用TypeScript，所以我们深入了解一下他是什么以及他和纯ES6的区别是什么。
TypeScript自己的网站上是这么定义他自己的：

> "一个有类型的JavaScript的子集，编译成纯JavaScript"

可能你看了这段描述之后还是摸不着头脑TypeScript到底是啥（虽然相对于it世界的茫茫宇宙，这段描述看起来相当简单）。实际上，在StackOverFlow上有一个帖子更详细的解释了TypeScript是什么--基本上，TypeScript为javascript增加了类型，类和接口。
通过TypeScript，你可以更严谨的去编程，就跟其他面向对象的语言(如Java，C#)一样。
JavaScript原本就不是设计用来做复杂应用的。虽然我们之前了解过JavaScript可以通过函数来实现仿类的行为，但是还是不够简洁。
但是，我之前说过ES6已经可以创建类了，为什么我们还是要用TypeScript？我看到一个Redditor说的很简单：

> "他叫做TypeScript，不叫ClassScript"

TypeScript始终提供了JavaScript的静态类型（意思是在编译时评估，与动态类型不同，是在运行时评估）。在TypeScript中使用类型是这样的：

```
function add(x: number, y :number):number {
    return x + y;
}
add('a', 'b'); // compiler error
```

以上代码陈述了x必须是number类型*(x : number)*，y必须是number类型*(y : number)*，add函数需要返回一个number类型的值*add():number*。因此，在这个例子中，我们将会收到报错，因为我们向一个只接受number类型的函数传入字符类型。这种制作大型应用的时候非常有用，添加了一层检查来预防bug。
如果你之前的Ionic 2代码的话：

```
export class MyApp {
    rootPage: any = HomePage;
    constructor(platform: Platform) {
        platform.ready().then(() => {

        });
    }
}
```

你可以发现一些TypeScript特性在发挥作用。以上代码告诉你*rootPage*可以是*any*类型，*any*是一个特殊类型，意思是可以是任何类型，*platform*是一个*Platform*类型。稍后，你将看到，对于依赖注入来讲，给事物进行类型限定是非常方便。

由于TypeScript是Ionic 2默认使用的，也许大多数人在使用的，所以本书专注于使用TypeScript。绝大部分ES6和TypeScript项目看起来非常像，在两者之间转换是非常简单的。

转译

转译的意思是将一种语言转换成另一种语言。为什么这个对我们来说很重要？基本上，EXMAScript 6提供给我们很多新鲜玩意，但是ES6只是个标准，还没有浏览器完全支持。我们得使用转译器将ES6转换成ES5（你现在在用的JavaScript）代码以协调浏览器。
在Ionic应用中，通过以下几个步骤来实现：

- 使用*ionic serve*运行应用
- **app**文件夹内的所有代码都转译到有效的ES5代码
- 将会创建一个打包好的JavaScript代码包并运行之

你不用去关心这个流程，因为Ionic自动帮你完成他。

网络组件

网络组件是Angular 2里面的大玩意，在Angualar 1中不是很好用。网络组件也不是Angular独有的，他们已经逐步成为了网络上的标准，用来创建模块，自包含，作为一小段代码很容易插入到页面中（跟WordPress中的Widget有点类似）。

> “In a nutshell, they allow us to bundle markup and styles into custom HTML elements.” - Rob Dodson

Rob Dodson写的这篇关于网络组件的文章，解释了他们是怎么工作的，以及他们背后的理念是什么。同时，他提供一个很好的例子，我觉得一步到位的解释了为何网络组件非常有用。
一般来讲，如果你要添加一个图片滑动作为网络组件，HTML应该是这样的：

```
<img-slider>
    <img src="images/sunset.jpg" alt="a dramatic sunset">
    <img src="images/arch.jpg" alt="a rock arch">
    <img src="images/grooves.jpg" alt="some neat grooves">
    <img src="images/rock.jpg" alt="an interesting rock">
</img-slider>
```

而不是这样的（没用到网络组件）：

```
<div id="slider">
    <input checked="" type="radio" name="slider" id="slide1" selected="false">
    <input type="radio" name="slider" id="slide2" selected="false">
    <input type="radio" name="slider" id="slide3" selected="false">
    <input type="radio" name="slider" id="slide4" selected="false">
    <div id="slides">
        <div id="overflow">
            <div class="inner">
                <img src="images//rock.jpg">
                <img src="images/grooves.jpg">
                <img src="images/arch.jpg">
                <img src="images/sunset.jpg">
            </div>
        </div>
    </div>
    <label for="slide1"></label>
    <label for="slide2"></label>
    <label for="slide3"></label>
    <label for="slide4"></label>
</div>
```

在不远的将来，你不需要去下载jQuery插件，然后复制粘贴一堆的HTML到你的文档中，你只需要导入这个网络组件，然后像上面那样子添加一个简单的图片滑动代码，就可以用来。网络组件非常有趣，如果想要知道他们是怎么工作的（例如Shadow Dom和Shadow Boundaries），那么，我强烈推荐你读一遍： Rob Dodson关于网络组件这个帖子。

译者于2017-5-2日看完全书，基本跑通所有代码，除IBM PouchDB部分外。

译者的 **ionic info** 如下：

```
cordova cli:6.5.0
Ionic CLI Version:2.2.3
Ionic App Lib Version:2.2.1
os:Windows 7
Node Version:v6.9.4
```

当前版本与作者成书版本有所不同。
例如作者成书的时候plugin皆以new关键字新建实例来使用，而译者学习的时候使用的版本则是以service的方式直接注入到需要用到类的constructor中去使用；
例如，以下代码来自-- 第七课：整合本地通知与社交分享：

```
import { Component } from '@angular/core';
import { Platform } from 'ionic-angular';
import { HomePage } from '../pages/home/home';
import { LocalNotifications } from 'ionic-native';

@Component({
    template: `<ion-nav [root]="rootPage"></ion-nav>`
})
export class MyApp {
    rootPage = HomePage;

    constructor(platform: Platform) {
        platform.ready().then(() => {
            if(platform.is('cordova')){
                LocalNotifications.isScheduled(1).then( (scheduled) => {
                    if(!scheduled){
                        let firstNotificationTime = new Date();
                        firstNotificationTime.setHours(firstNotificationTime.getHours(
)+24);

                        LocalNotifications.schedule({
                            id: 1,
                            title: 'Snapaday',
                            text: 'Have you taken your snap today?',
                            at: firstNotificationTime,
                            every: 'day'
                        });
                    }
                });
            }
        });
    }
}
```

但是译者使用的最新版本里面应该是这样去用的：

```
import { Component } from '@angular/core';
import { Platform } from 'ionic-angular';
import { HomePage } from '../pages/home/home';
import { LocalNotifications } from '@ionic-native/local-notifications';

@Component({
    template: `<ion-nav [root]="rootPage"></ion-nav>`
})
export class MyApp {
    rootPage = HomePage;

    constructor(platform: Platform,public localNotification:LocalNotification) {
        platform.ready().then(() => {
            if(platform.is('cordova')){
                this.localNotification.isScheduled(1).then( (scheduled) => {
                    if(!scheduled){
                        let firstNotificationTime = new Date();
                        firstNotificationTime.setHours(firstNotificationTime.getHours(
)+24);

                        this.localNotification.schedule({
                            id: 1,
                            title: 'Snapaday',
                            text: 'Have you taken your snap today?',
                            at: firstNotificationTime,
                            every: 'day'
                        });
                    }
                });
            }
        });
    }
}
```

注意LocalNotification的使用方式以及导入方式。

又如，所有的native api都已分包，具体可以参考官方文档用法。

用到的一些名词：（更新与2017/5/15）

- default 默认
- handler 操作器（由于昨天读文章看到一般翻译为：句柄，所以才会有这里的名词约定）
- method：方法
- function：函数
- native：本地，从3.4开始翻译为【本机】，前面的翻译有空在核对
- local：本地（会有备注和上面的进行区分）
- functionality：功能

- feature：特性
- item：条目，列表项
- provider：提供者
- property,attribute：属性（很难区分）
- helper：助手，助理
- instance：实例
- object：对象
- build：构建
- subscribe：订阅（针对Observable）
- splash screen：闪屏，启动画面（有更好的翻译请告诉我）

译者当前学习完后的可用代码在此，仅限与代码部分：

source_code:ionic v3.1.1

第二章

Ionic 2 基础

第一课：生成一个Ionic2应用

我们已经讲了不少内容了，现在我们需要知道Ionic 2是干什么的，为什么要对他做这么多更改。带着这个疑问，我们开始学习如何使用Ionic 2。

安装Ionic

在使用Ionic 2制作音乐之前，我们需要装好所有需要用到的东西。用的是Mac还是PC不重要，我们都可以用来学完这本书生产一个iOS和Android应用提交到App商店。
重要：如果你电脑上已经装了Ionic 1那么你可以直接跳到下一部分。你只需要运行*npm install -g ionic*或者*sudo npm install -g ionic*来设置Ionic 2需要的任何东西。如果你想继续使用Ionic 1的话，用不着担心，更新之后，Ionic 1和Ionic 2你都可以用。
首先，你的装个Node.js。Node.js是一个用来制作快速，伸展性网络应用的平台，他可以用来做很多不同的事情。如果你不熟悉Node.js你也无须担心，我们基本上不怎么用的上-但是他是跑Ionic和安装其他包必备的。

> 访问以下网址安装**Node.js**:
https://nodejs.org/en/download/

一旦安装好了Node.js，你可以通过命令行去访问node包管理器了。
> 在终端运行如下命令安装**Ionic**和**Cordova**：

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

或者

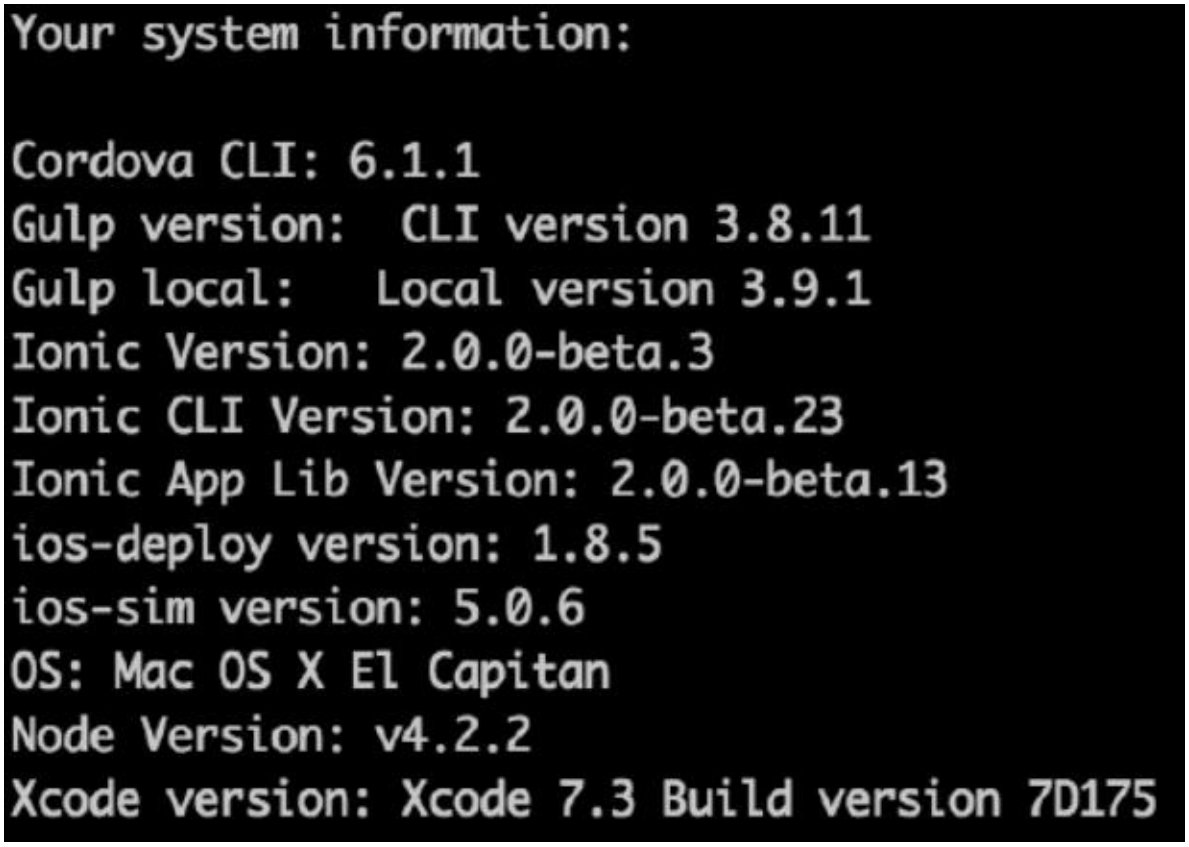
```
sudo npm install -g ionic cordova
```

如果没有设置好你的Android SDK的话，请参考如下方案去设置好：

- 在Mac上安装Android开发环境
- 在Window上安装Android开发环境

如果你是Mac机的话，那么你也要安装好XCode，因为你需要用他来构建应用和为应用签名。
iOS SDK的配置无须担心，因为XCode已经帮你摆平了，如果你没有Mac的话，那么你就没法设置了。（后续我们会讲到如何在没有Mac的情况下构建iOS应用）
现在你的开发环境应该设置好了。通过以下命令，查看你是否成功安装Ionic CLI（Command Line Interface，命令行交互界面）：''' ionic -v ''' 你也可以通过在你的ionic项目内运行: '''

ionic info "" 来查看更多信息。
以下是我写作本书的时候的输出信息：



如果你在安装ionic或者生成新项目的时候发生错误，先保证下你是否安装了最新（当前版本）的Node。安装好了最新的版本之后，先运行如下命令：

```
npm uninstall -g ionic npm cache clean
```

注意：Ionic框架和Ionic CLI是两个不同的东西。CLI是咱们刚装的，他通过命令行提供了一系列的工具帮助我们创建和管理Ionic项目。Ionic CLI负责下载实际的Ionic框架到每个你创建的项目中。

生成第一个Ionic项目

Ionic装好后，生成应用特别简单。你只需要简单的运行*ionic start*命令就可以创建一个新项目，包括一些模板代码和文件。
> 运行如下命令生成一个新的**Ionic**应用：

```
ionic start MyFirstApp blank --v2
```

以上命令会生成一个名为'MyFirstApp'的应用，使用的是'blank'模板。Ionic有一些内置的模板，上面我们用的上'blank'，我们也可以用其他的：

```
ionic start MyFirstApp sidemenu --v2
```

或者

```
ionic start MyFirstApp tutorial --v2
```

又或者运行默认命令：

```
ionic start MyFirstApp --v2
```

默认的是一个标签页应用模板。记住，想要创建Ionic 2应用的话就必须在后面加'--v2'。如果不加的话就会创建Ionic 1项目了。
注意：当前所有Ionic 2项目默认使用TypeScript。由于TypeScript是一个ES6的扩展，所以可以在TypeScript项目里面直接使用ES6，但是所有的Javascript必须是.ts，而不是.js。
下面我们开始研究枯燥的blank模板项目。项目创建成功之后，立刻将项目作为当前目录，这样我们就可以对他做更多事情了。
> 运行如下命令转到**Ionic**项目目录

```
cd MyFirstApp
```

如果对命令行或者终端不熟悉的话，先考虑读下这个。虽然内容是针对Ionic 1的，但是里面有关于命令行界面工作原理。

添加平台

终于，我们要开始使用Cordova制作我们的应用了（实际上Ionic CLI生成的应用是一个Cordova应用），首先我们得添加我们的目标平台。如果要添加Android平台的话，那么运行这个命令：

```
ionic platform add android
```

添加iOS平台的话，运行：

```
ionic platform add ios
```

如果你制作的是这两个平台应用的话，那么你都应该运行一下。这样你的项目就可以构建对应平台的应用了。 在这里我简单解释一下，我们大部分的代码将会在**app**文件夹内，同时你也可以在你的项目下找到一个名为**platforms**的文件夹--对应平台的相关配置都是存放在此处

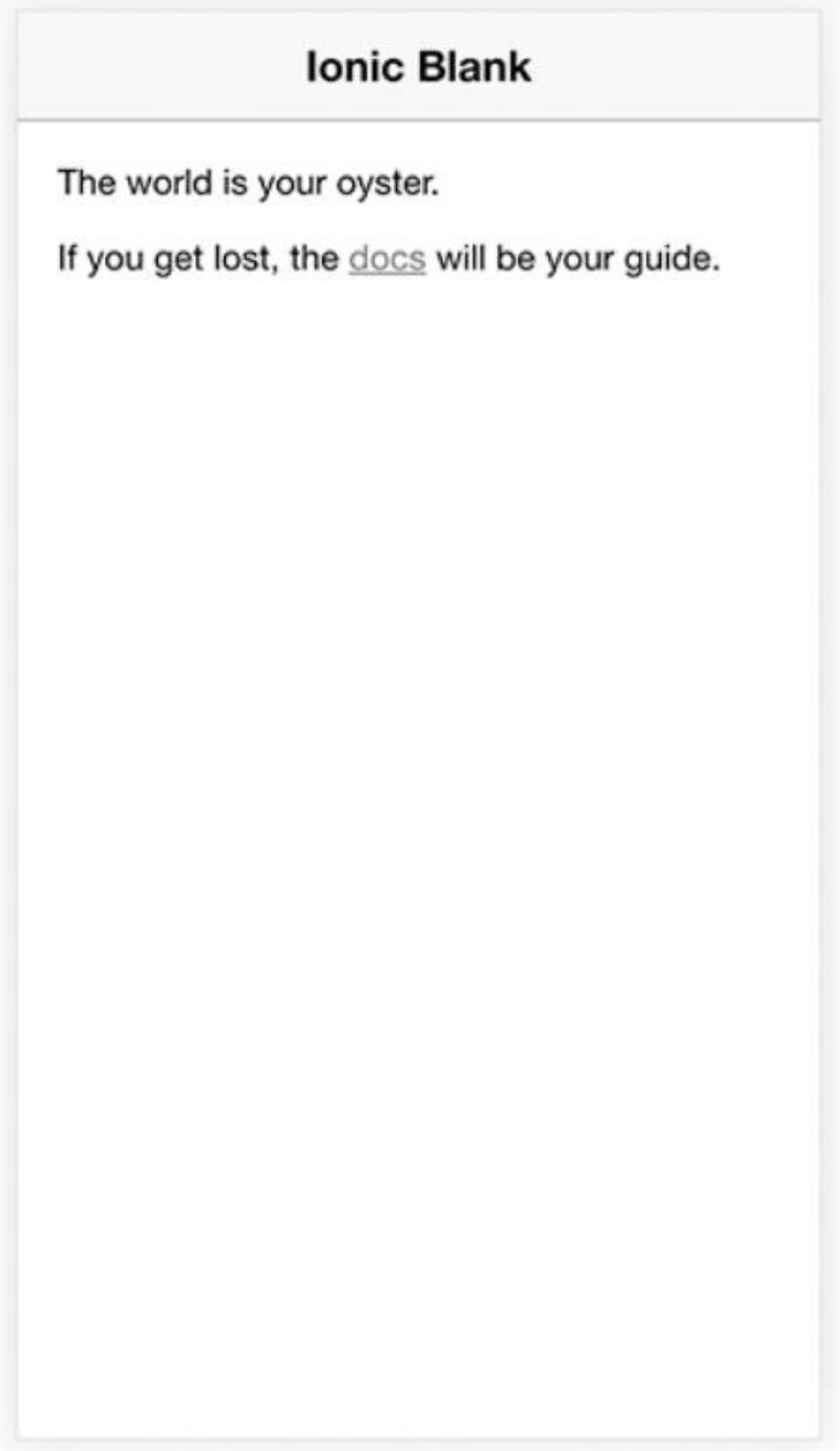
的。后续我们会集体讲一遍。

运行应用

HTML5移动应用的美好之处在于你可以一边开发一边做浏览器中查看他。但是如果你直接利用浏览器打开**index.html**文件的话，估计你遭遇到不开心的事情。
Ionic项目是需要到服务端上运行的--意思是你不能直接访问这些文件，也不是意味着你需要把他放到互联网上去，你可以将一个完全自容的Ionic应用部署到应用商店去（这个我们后面会学到）。幸运的是，Ionic提供一个了一个简单的在本地网络服务器同步开发和测试的方法。
> 运行如下命令可以在浏览器中预览你的应用：

```
ionic serve
```

这个命令会打开一个新的浏览器，浏览器会打开你当前应用在本地服务器上运行的地址。目前，他看起来应该是这样的：



这个命令不仅可以给你查看你的应用，当你代码改动的时候，他会实时更新显示。让你编辑或者保存任何文件的时候，你不用在浏览器中重新刷新页面就可以看到更改自动映射到浏览器中了。

停止此进程请使用：

```
Ctrl + C
```

同时请记住，运行了*ionic serve*后不能运行其他Ionic CLI命令来。你需要通过 Ctrl + C 终止此命令然后运行指定的命令。（译者注：Windows上可以重新开一个命令行来运行）

更新应用

有时候你可能需要更新到新的Ionic版本。更新项目中的ionic版本最简单的方法是先更新Ionic CLI：

```
npm install -g ionic
```

或者

```
sudo npm install -g ionic
```

然后，更新项目中的package.json，你应该可以在其中看到这样的一部分：

```
"dependencies": {
    "@angular/common": "^2.0.0",
    "@angular/compiler": "^2.0.0",
    "@angular/compiler-cli": "0.6.2",
    "@angular/core": "^2.0.0",
    "@angular/forms": "^2.0.0",
    "@angular/http": "^2.0.0",
    "@angular/platform-browser": "^2.0.0",
    "@angular/platform-browser-dynamic": "^2.0.0",
    "@angular/platform-server": "^2.0.0",
    "@ionic/storage": "^1.0.3",
    "ionic-angular": "^2.0.0-rc.1",
    "ionic-native": "^2.2.3",
    "ionicons": "^3.0.0",
    "rxjs": "5.0.0-beta.12",
    "zone.js": "^0.6.21"
},
"devDependencies": {
    "@ionic/app-scripts": "^0.0.33",
    "typescript": "^2.0.3"
},
```

只要改掉*ionic-angular*的版本号到最新的版本号，然后在项目文件夹中运行：

```
npm install
```

就可以了。这个命令将自动抓取最新版的框架并添加到你的项目中来。
重要：记住，如果需要更新**package.json**其他依赖库到最新版本的话也是一样操作的。
在新版本发布之后，先阅读更新日志查看是否有任何重大变更，因为如果有的话你就有可能需要修改部分项目代码了。
通常做法是利用更新后的Ionic CLI新建一个项目，然后将你在个人代码覆盖进去。如果不想这么做的话，一定要确保仔细阅读了更新日志，然后相应的更新你的依赖库和代码。随着Ionic 2越来越稳定，后续变动其实也不会很大，所以这也不会是个太大的问题。

第二课：剖析Ionic2项目

现在我们知道了如何安装Ionic 2以及如何生成项目，接下来我打算讲解一下新建项目里包含的那些文件和文件夹的内容。当你创建一个blank Ionic 2应用的时候，你的文件夹结构看起来应该是这样的：

乍眼一看，好多东西啊 -- 但是需要你去操心的真心没那么多，在简单解释之后你就会明白的。我们将会谈论每个东西是干啥的，但是我还是会从最重要的部分（也就是做项目的时候你需要改动的部分）开始详细讲解，然后顺道讲解一下其他次要部分。

重要的文件和目录

这些文件和目录是你需要频繁用到的，所以明白他们所扮演的角色对你来说非常重要。幸运的是，他们没多少！

src

这里是所有动作的发生地。**src**文件夹内默认会包含如下内容：

- **app**文件夹
- **pages**文件夹
- **theme**文件夹
- **assets**文件夹
- **index.html**文件夹

pages文件夹包含了应用所有页面组件。如果打开**pages**文件夹，你可以看到一个名为**home**的文件夹。他是一个组件（**component**），由一个类定义（home.ts），一个模板（home.html），一些样式定义（home.scss）。任何应用的其他页面都会在此有一个对应的文件夹（实际上我们可以交给Ionic CLI帮我们自动生成这些文件），因此，你的项目可能也有**login**，**intro**，**checkout**，**about**等等其他文件夹。
theme文件夹包含所有的**.scss**文件，这些文件用于为应用定义应用级范围的样式。这里面包含了一个共享变量的文件，你可以覆写这些变量，一个全局文件用于存储整个项目的通用样式。后续有一整个部分来讲解如何给Ionic 2应用制定主题，所以我们后续详细探讨。
app文件夹包含了应用的根组件**root component**，也就是**app.module.ts**。再一次，根组件将会在后续章节详细探讨，此处暂时略过，实际上，这是整个应用的'起点'。这里会有一个**app.module.ts**文件，此处我们用到了Angular 2的*@NgModule*以用来设置应用的所以依赖，例如我们用到的组件和服务。最后，你会看到*main.dev.ts*文件和*main.prod.ts*文件，这是应用的引导启动。*dev*文件用在开发环境中，*prod*用在生产环境中。
引导流程只是普通的设置应用的根组件，根组件是第一个创建的组件然后一直使用到最后。
assets目录存储一些应用会用到的静态资源，例如图片，JSON文件。这些资源会在应用编译的时候拷贝到编译文件夹（所以，将素材放在这里非常重要，而不是放到*www/assets*文件夹）。你可以在此处建立一个文件夹名为*images*，然后在项目里这样引用*assets/images/myImage.png*。
*src/index.html*虽然不经常改动，但是你还是可以去改的，与*assets*一样，他会被拷贝到build目录下面去。
和你的应用默认包含的文件夹一样，当你开始编译你的应用的时候，你将会在这里看到一些其他的文件夹，这个我们后续在讨论。
重要：这里非常重要，如果你还是理解的话请多读几遍。在介绍部分我们说到了webpack，转译，以及其他一些好腻害的ES6特性 -- 重要的一点是需要记住我们用到的ES6和TypeScript特性浏览器不一定支持，所以我们需要将他们'转译'到ES5代码。当你运行或者编译应用的时候，Ionic将会帮你自动打包好app文件夹下面的所有内容，执行他需要执行的效果，然后全部输出到**www**文件夹。
当你在浏览器中预览应用的时候，你其实看到的是**www**文件夹里面打包的版本，而不是

你**app**文件夹里面的内容。同理，当你打包iOS和Android发行版的时候（后续讨论），也是用的**www**文件夹，而不是**app**文件夹。不要编辑**WWW**文件夹里面的代码 -- 你在里面做的变动在编译的时候会被自动覆盖。
使用Angular 2和新ES6语法以为这项目结构和编译流程会稍微复杂，幸运的是Ionic都帮我们搞定了。所以无需担心太多，只需要记在脑海即可。

config.xml

这个文件不需要改动频繁，但是这个文档非常重要。**config.xml**文件用于高速Cordova如何将你的项目编译到iOS和Android。你可能需要提供一些重要的配置信息，这个我们稍后讨论，你也可以在此定义一些首选项（例如闪屏是否需要自动隐藏，应用是否仅限竖屏，等等）。
你只有当你的应用跑着眞机上的时候才需要关心这个文件。

不那么重要的东西

很明显，Ionic项目中每个事物的存在都有其存在的道理。但是其中的一些用于更高级用途，这些你可能不需要关心；有一些可能是纯粹的配置，你永远都不要去改动。

配置文件

如果看一下你生成的项目的话你会看到一大堆的配置文件。
除了上面讲过了的 **config.xml** 之外，你可以完全忽略这些文件。唯一你可能会编辑的文件是 **package.json** ，他可以用来更新Ionic的版本。

resources

这个文件纯粹是用来防止你编译应用的时候需要用到的闪屏和图标。后续我会给你展示一个更简单的方法利用Ionic CLI来生成这些资源。

hooks

hooks用于应用编译流程的一部分，你可以在这里添加自定义的脚本来跟编译流程的不同部分挂钩。初学者不太需要去碰触着部分的内容，但是如果你想开发更复杂的工作流，例如版本相关或者应用部署相关，你可以在此处创建一些'hooks'。

node_modules

这是另一个你不需要去碰的文件夹，但是这里是所有好玩意儿存放的地方。如果浏览一下的话你会发现**ionic-angular**，**angular2**，以及**ionicons**都在这里。这里是所有项目依赖库的存放点（包括Ionic框架本身）。

第三课：Ionic CLI命令

Ionic CLI是一个非常强大的工具 -- 我们已经用他生成过一个新项目然后在浏览器中显示，但是还是有很多命令需要你去了解，那么我们现在开始了解这些命令吧。这虽然不是一个巨细无遗的列表，但是会覆盖你需要经常用到的命令。
我现在将列出一些命令和他们做什么，由于一些命令可以接受多个不同的参数，因此我将使用以下语法：

```
ionic command [option1|option2]
```

代替：

```
ionic command option1
```

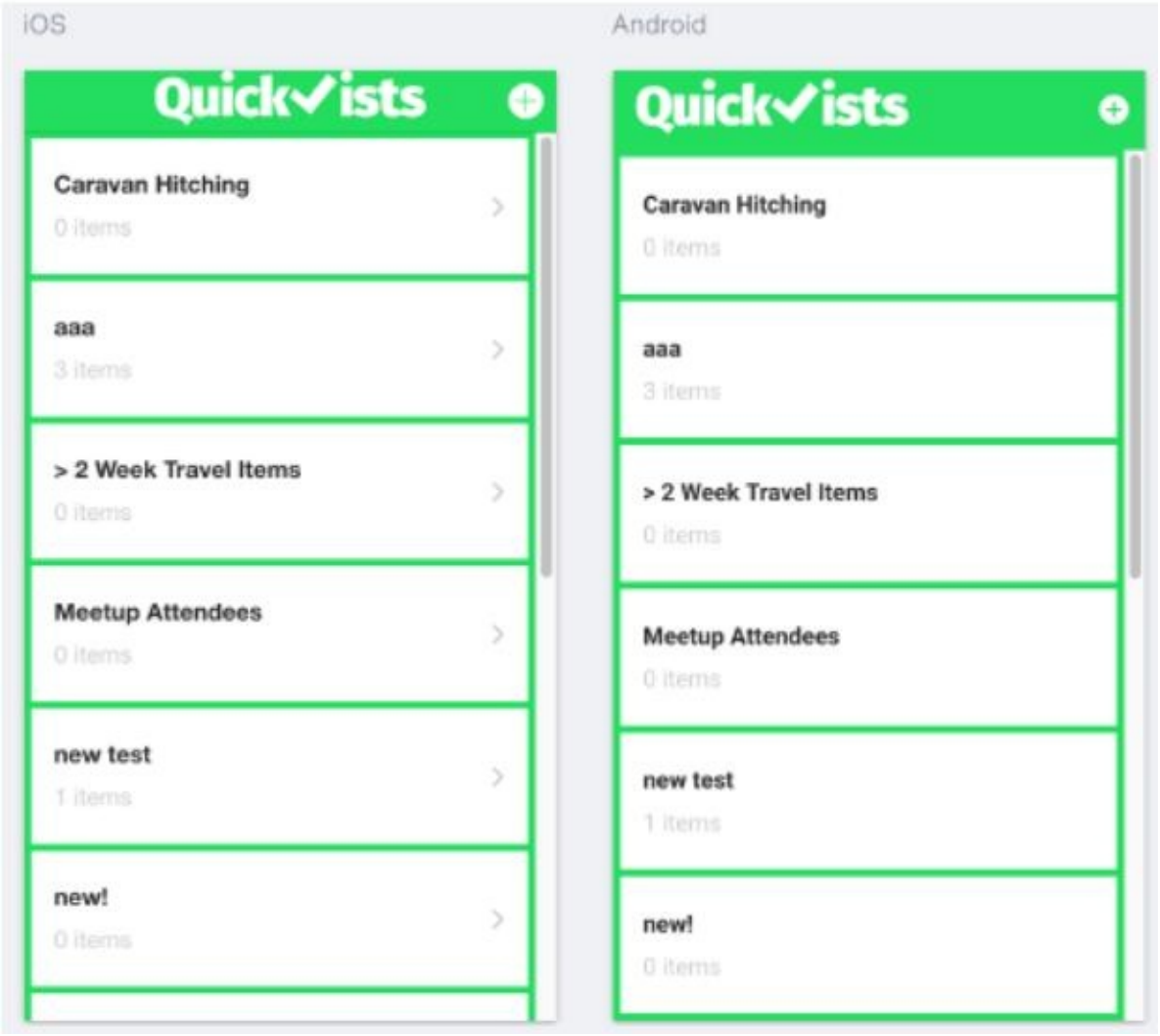
和：

```
ionic command option2
```

为整洁明了起见，我们现在就开始：

```
ionic serve -l
```

你已经见识过*ionic serve*了，他会在浏览器中启动你的应用并实时重新加载，但是这个命令或启动Ionic Lab，看起来将是这样的：



很直观的并排给你展示你的应用中iOS和Android上的显示。

```
ionic platform add [ios|android]
```

这个命令用来添加你想要编译的目标平台。

```
ionic plugin add [plugin]
```

这个命令用来给你的项目添加Cordova插件，只要提供插件名就可以了。

```
ionic build [ios|android]
```

如果你准备编译你的应用为iOS和Android应用的话，使用这个命令。真正想把他打包到你的移动设备上使用然后提交到App Store的话会稍微复杂一些，后续会讲到。

```
ionic run [ios|android]
```

如果想在真机上运行你的应用，可以通过这个口令将应用直接部署到你的设备上去。详细请参考本书‘测试&调试’部分。

```
ionic emulate [ios|android]
```

这个命令不需要将应用部署到真实元件，而是在本机启动模拟器来运行应用。

```
ionig g [page|component|directive|pipe|provider|tabs]
```

这是一个生产命令，是我最喜欢的命令之一。用他来生成非常节省时间。我之前说过，Ionic项目里面会有大量的组件，例如**home**页面。你可以手动去创建一个新的文件夹然后在其中添加需要的文件以创建一个组件，或者你可以使用*ionic generate*命令，来自动帮你生成对应的模板内容。这个命令不仅可以用来生成页面，还可以用于生成同城的组件，指令，管道，提供者以及页签。建议在添加新组件的时候用它来处理。

```
ionic plugin list
```

展示项目安装的插件列表。

```
ionic plugin rm [plugin]
```

移除已安装的插件，需要提供插件名。

```
ionic platform rm [android|ios]
```

移除之前添加的平台。
之前说过，这不是一个完整的清单，但是它涵盖了所有会经常用到的命令。那些基本不会用到的命令此处没有涉及。

第四课：装饰器

Ionic 2应用里每个类（后续会讲到）都会有一个装饰器。装饰器看起来是这样子的：

```
@Component({
    someThing: 'somevalue',
    someOtherThing: [Some, Other, Values]
})
```

他们看起来好奇怪，但是他们扮演着至关重要的角色。在Ionic 2中他们的角色是给你定义的类*class*提供元数据*metadata*，他们通常都是一屁股坐在类头上的，就像这样：

```
@Decorator({
    /*meta data goes here*/
})

export class MyClass {
    /*class stuff goes here*/
}
```

这也是你唯一可能看到装饰器的地方，他们纯粹是用来给类添加一些额外信息的（即，他们'装饰'类）。我们接下来聊聊为什么Ionic 2应用中需要用到他，以及怎么去用。
装饰器的名字本身就很有用，以下是在Ionic 2应用中用到的：

- @Component
- @Pipe
- @Directive

可以给装饰器提供一个对象以提供更多我们想要知道的信息。下面在应用中最普通常见的：

```
@Component({
    selector: 'home-page',
    templateUrl: 'home.html'
})

export class HomePage {

}
```

这样类就知道该从哪里拿到他的模板。如果你的模板超级简单，那么你都可以不用建立另外的模板文件，而是直接传入：

```
@Component({
    template: `<p>Howdy!</p>`
})
export class HowdyPage {

}
```

有些甚至喜欢在 *template* 上定义大的模板。由于ES6支持使用反引号（上面包住模板内容的东西）来定义多行字符串，这也是定义大型模板的一个可选方案（而不是丑陋的把大量字符串串联起来）。
现在，我们了解了什么是装饰器以及他用来干什么，接下来我们来看点特别的。

Ionic 2应用中的通用装饰器

我们能用到的装饰器也不是很多。他们的最终目的基本都是简单的描述我们创建的类似什么，这样他就知道他需要导入什么来正常工作。
我们来了解一下我们主要会用到的装饰器，以及他们分别对应的角色。我们目前只是关注装饰器，下一部分才会讨论如何通过类做一些有用的事情。

@Component

我觉得在Ionic 2应用中组件 *component*的任务比较让人困惑。就像我说的，咱们的原因都是由大量的相互绑定的组件组成。这些组件都包含在咱们的**app**文件夹内，看起来大概是这个样子的：
home

- home.ts
- home.html
- home.scss

@Component不是Ionic 2特有的，他在Angular 2里面经常用到。Ionic 2提供的大部分功能都是通过左键完成的。例如，在Ionic 2中，如果你想创建一个搜索条，你可以使用Ionic 2提供的这个组件：

```
<ion-searchbar></ion-searchbar>
```

你只需要简单的将这个自定义的变迁添加到你的模板就可以了。Ionic 2虽然提供了大量的组件，你还是可以创建自定义的组件的，自定义的组件的装饰器看起来大概是这样子的：

```
@Component({
    selector: 'my-cool-component'
})
```

这样一来你就可以直接在模板中这样子用了：

```
<my-cool-component></my-cool-component>
```

注意：技术层面上来讲组件必要有一个类定义和一个模板。像管道和数据提供者这样的组件不会在屏幕上显示所以不会有相关的模板，他们仅用于提供其他的功能。虽然这些在技术层面上不能称之为组件，但是你还是需要经常用到的。

@Directive

@Directive装饰品允许你创建自定义的指令。这些装饰器一般看起来是这样子的：

```
@Directive({
    selector: '[my-selector]'
})
```

然后在模板中你可以通过在元素是添加这个选择器来触发指令的行为。

```
<some-element my-selector></some-element>
```

什么时候用**@Component**什么时候用**@Directive**可能会有点容易混淆，因为他们比较像。最简单的方法是如果你想要改变存在的元素的行为，那么请使用**directive** 指令，如果你想创建一个新的组件，那么请使用**component** 组件。

@Pipe

@Pipe用来创建自定义的管道以过滤需要展示给用户的数据。他的装饰器看起来是这样的：

```
@Pipe({
    name: 'myPipe'
})
```

然后你就可以在模板里面这样去使用：

```
    <p>{{someString | myPipe}}</p>
```

这样的话 *someString* 在展示给用户之前会先经过 *myPipe* 过滤。

@Injectable

@Injectable用来创建供类使用的服务。例如，我们在制作应用的时候，经常会有用于获取数据保存数据的数据服务 **Data Service**就是一个典型的此类服务。无需在类中手动编写这些代码，你只需要在要用到的类中注入此数据服务即可，然后再类中调用数据服务的辅助方法。当然，数据服务不是唯一可做的，你可以利用他做任何你想要的事情。
@Injectable通常就是一个**@Injectable**装饰器蹲在一个类上面：

```
@Injectable()
export class DataService {

}
```

重要：请记住，你想做Ionic 2中使用任何东西之前，请先导入（关于导入后续的章节我详细涉及）。像管道，指令，注入（服务）和组件这些东西，他们不但要导入，同时也需要到**app.module.ts**中声明。这些在制作应用范例的时候会详细解释。

总结

关于装饰器最重要的需要记住的事情是：没多少需要去死记硬背的。装饰器非常强大，你也可以试一试一些看起来比较复杂的配置。随着你对Ionic 2的深入，你的装饰器也会越来越复杂，但是，在开始阶段，你的装饰器主体大概这样就够了：

```
@Component({
    selector: 'home-page',
    templateUrl: 'home.html'
})
```

很多人第一眼看到装饰器可能就已经放弃了，因为第一眼看这其实比较奇怪，比较他本身吓人。下一课我们会学习装饰器的犯罪搭档：类。类定义实际上我我们完成所有工作的地方，记住，装饰器在顶部，提供一点点额外的信息。

第五课：类

上一个部分我们详细了解了什么是装饰器。概括起来就是类定义上面那一小段代码声明这个类是什么，这个类需要什么，以及这个类应当如何配置。现在，我们要来聊一聊类了。

类是什么？

基于你自己的编程经历，你有可能知道，也有可能不知道类是什么。所以，这里我先后退一步，解释一下在同一编程理念里面类似什么，因为类不是Ionic，Angular或者JavaScript独有的概念。
类是面向对象编程（OOP）里面使用的，他们本质上是作为对象的'蓝图'去使用的。你可以定义一个类，然后对他进行创建，实例化以及对象化。如果你对类一无所知，在继续进行课程之前请学习一些关于类的基本知识。那么，我们先来看一个简单的例子吧：

```
class Person {
    constructor(name, age){
        this.name = name;
        this.age = age;
    }
    setAge(age){
        this.age = age;
        return true;
    }
    getAge(){
        return this.age;
    }
    setName(name){
        this.name = name;
        return true;
    }
    getName(){
        return this.name
    }
    isOld(){
        return this.age > 50;
    }
}
```

这个类定义了一个**Person**对象。*constructor* 构造器在创建类的实例时候会运行，他接受两个值：name和age。这两个值用于设置类的成员变量，也就是 ***this***.*name* 和 ***this***.*age* 。 这些值可以在类定义里面的任何地方通过***this***关键字来访问。***this***关键字是当前范围 **scope**的引用，所以他的引用取决于你在哪里使用他，但是如果你在一个类里面使用（不是回调里面或者其

他将会改变范围的东西里面），那么他就是类本身了。
可以把你自己想象成***this***，你所在的物理世界是范围，考虑这个情况：如果你在旅馆的房间的话那么你的范围就是这个房间，如果你离开房间的话，那么你的范围就是旅馆里，如果你离开旅馆的话，那么你的范围就编程了世界（如果你想说国家的话也是可以的）。
如果你不熟悉这个关键字的话，推荐先阅读此文。
我们有自己的类定义，用于作为创建对象的蓝图，这样一来我们就可以像这样创建一个新的**Person**对象：

```
var john = new Person('John', 32);
```

这里我传入的两个参数将会传递到**Person**的***constructor***用于设置成员变量。如果我们现在运行如下代码：

```
console.log(john.getName());
```

John的名字将会输出到控制台。同理我们也可以调用*getAge*函数来得到他的年纪，我们也可以通过*set*函数来更改他的名字或者年龄。*Getter*和*Setter*在类里面非常常见，同时我们也定义了一个非常有趣的函数 *isOld*。这个函数在**Person**年龄大于50的时候返回*true*，咱们的Johnm看起来年纪是没有这么大的。
可能最需要记住的概念是类就是一个‘蓝图’，对象可以看到是类的一个独立副本。所以我们可以创建同一个类的多个实例，例如：

```
var john = new Person('John', 32);
var louise = new Person('Louise', 28);
var david = new Person('David', 52);

console.log(john.isOld());
console.log(louise.isOld());
console.log(david.isOld());
```

在以上代码中John，Louise和David都是**Person**类的独立个体对象，他们的值都是分开维护的。如果运行以上代码的话，只有David会返回（他可能比较老，但是可以肯定的是他很正直）**true**。

Ionic 2里面的类

现在，我们知道类是什么了，但是为什么他们会突然出现在Ionic 2和Angular 2里面呢？我们之前略表过，类是ES5规格的JavaScript的新特性。这个新特性当然是大受欢迎的，因为这个是编程界使用最广泛的模式，实际上JavaScript应用早就开始使用这种方式了，ES6只是把他

正式化了。
ES6之前都是通过函数 **functions** 来达到类型类的结构（估计现在大部分人还是这么做的，因为现在基本还是ES5的天下）。大概是这样的：

```
var Person = function (name, age) {
    this.name = name;
    this.age = age;
};
Person.prototype.isOld = function() {
    return this.age > 50;
};
var david = new Person('david', 52);
console.log(david.isOld());
```

看起来稍微有一点点不同，但是最终结果还是基本一致的。由于ES6有了*class*关键字，我么现在可以用正'正常'的途径来做了。
我们看一下Ionic 2里面类大概是怎样的：

```
import { Component } from '@angular/core';
import { NavController } from 'ionic-angular';
import { SomePage } from '../pages/some-page/some-page';

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

    }
}
```

可能你第一眼就注意到了*import*语句了。你需要在类中用到的东西都需要导入。因此，我们从 **@angular/core** 导入 **Component** 这样我们就可以去使用 **@Component** 装饰器，以及 Ionic 库里的 **NavController** 用来控制页面导航。
同时我们也导入了我们自己创建的 **SomePage**。路径都是跟随项目目录结构的，在这个例子中 **SomePage** 组件定义在当前文件的上一层目录的 **pages** 文件夹里。导入路径是链接到 **.ts** 文件里的类，但是不需要加上 **.ts** 扩展名。
现在我们有了装饰器，在装饰器里面我们定义了他的选择器（即，这个组件在DOM中的名字，*</home-page>*）和他的模板。
定义完装饰器之后，我们终于达到了类本体了。我们注意到前面有一个 **export** 关键字，例如：

```
export class HomePage {

}
```

export 关键字和 **import** 关键字是串联的，我们想要在别的地方 **import** 类的话，我们得先 **export**。我们最后讨论的是凑早期。我们已经大概的探讨了一下构造器在类中扮演的角色，这里也不例外：构造器里面的代码会在类实例化的时候运行。

此处我们不止需要知道这个。在Ionic 2中，类里面需要用到的服务都需要注入到构造器中，看起来将会是这样的：

```
constructor(platform: Platform, nav: NavController) {
    platform.ready().then(() => {

    });
}
```

例子中我们需要利用 **Platform** 服务来检测设备准备好的时机，所以我们将他注入构造器，然后在构造器中用它。
在此处我们不需要在构造器以外的地方用到 **platform**，但是在大部分的情况下，你需要在其他地方用到注入的服务。所以，为了在类里其他函数里可以访问到这些服务，我们必须将它设为成员变量。代码应该是这样的：

```
import { Component } from '@angular/core';
import { Platform } from 'ionic-angular';
@Component({
    selector: 'home-page',
    templateUrl: 'home.html'
})
export class HomePage {
    someMemberVariable: any = “hey”!;
    constructor(platform: Platform) {
        this.platform = platform
    }
    someOtherMethod(){
        this.platform.ready().then(() =>{

        });
    }
}
```

或者在TypeScript里面我们可以利用***public***关键字自动为他创建成员变量引用，如下：

```
import { Component } from '@angular/core';
import { Platform } from 'ionic-angular';
@Component({
    selector: 'home-page',
    templateUrl: 'home.html'
})
export class HomePage {
    someMemberVariable: any = “hey”!;
    constructor(public platform: Platform) {

    }
    someOtherMethod(){
        this.platform.ready().then(() =>{

        });
    }
}
```

同时请注意任何定义在构造器上面的变量，如此处的***someMemberVariable***，都会自动设置为成员变量。所以，在本例中，我们可以随处使用***this***.*someMemberVariable* 来访问此变量。任何你想要用到的服务都需要注入（有的需要设置为成员变量）到构造器，任何需要用到的成员变量都需要定义在构造器上方。如果，当前这个观念对你来说有点迷糊，后续通过一些例子我们应该就会有感觉的。
现在，我们可以通过 ***this***.*platform* 在任何地方访问 **platform**。如果我们没有设置这个成员变量然后通过调用 *someOtherMethod* 去访问 **platform** 的时候，将没啥用。

创建一个页面

你的应用永远会有大部分是页面 -- 任何你想单独展示屏幕都将作为一个单独的 **Page** 定义。对之前而言，有一个特殊的装饰来完成，现在我们只用常规的 **@Component**。
我们之前讨论过，页面的类看起来应该是这样的：

```
import { Component } from '@angular/core';
@Component({
    selector: 'home-page',
    templateUrl: 'home.html'
})

export class MyPage {
    constructor() {

    }
}
```

这个装饰器里引用的模板大概是这样的：

```
<ion-header>
    <ion-navbar>
        <ion-title>
            My Page
        </ion-title>
    </ion-navbar>
</ion-header>
<ion-content>
    <ion-list>
        <ion-item>I</ion-item>
        <ion-item>Am</ion-item>
        <ion-item>A</ion-item>
        <ion-item>List</ion-item>
    </ion-list>
</ion-content>
```

模板文件组成了用户所见的东西（后续会更详细讨论模板）。模板文件和类协同工作：类定义了展示什么模板给用户，模板可以使用类里的数据和函数。
我们讲完了一个 **Page** 类的基本结构是什么样子的，以及 构造器 函数是干什么的，但是你也可以添加其他你的页面需要用到的函数，例如：

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

@Component({
    selector: 'home-page',
    templateUrl: 'home.html'
})

export class MyPage {
    constructor() {
        //this runs immediately
    }

    someMethod(){
        //this only runs when called
    }

    someOtherMethod(){
        //this only runs when called
    }
}
```

你可以在构造器里面调用这些方法，或者你可以通过用户在模板中点击一个按钮来触发。这些附加方法可以添加到任何其他的类，他不是针对页面才能拥有的。稍后会详细讲到这些，目前我们只需要理解不同类类型之间结构的不同以及他们的作用就可以了。

创建一个组件

普通组件的代码看起来跟页面差不多（记住，页面就是一个组件）。当创建有组件的页面的时候，我们使用Ionic内置的导航来处理他们的显示。页面是一个霸占整个屏幕（即，用户的'视窗'）的组件，但是一个组件允许创建你自定义的元素，然后用于插入到模板。有可能你需要创建一个自定义的日期选择器插入到页面中去，或者一个展示随机鸡汤文的框 -- 针对这样的需求你都可以创建一个自定义的 组件。
同样组件的类定义看起来差不多都是这样：

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

@Component({
    selector: 'my-component',
    templateUrl: 'my-component.html'
})

export class MyComponent {

    text: any;

    constructor() {
        this.text = 'Hello World';
    }
}
```

实际上此处组件唯一的不用是它指定了一个 **selector**。这就是你在将组件插入模板用到的名字。即：

```
<my-component></my-component>
```

在考虑使用他之前，我们先看看组件的模板。这跟页面的模板没有任何区别。我们引用的模板文件名为 **my-component.html** 文件的内容是这样子的：

```
<div>
    {{text}}
</div>
```

就跟页面一样，我们可以引用类定义里面存储的任何数据（函数也可以）。在这个模板中，我们的组件的工作是将下面内容渲染到DOM里去：

```
<div>Hello World</div>
```

确实很枯燥，但是你可以利用这个功能做很多有趣的，可重用的东西。现在，我们来看看如果在页面中使用这个组件。
你需要导入他然后加入到 **app.module.ts** 文件中：

```
@NgModule({
    declarations: [
        MyApp,
        HomePage,
        MyComponent
    ],
    imports: [
        IonicModule.forRoot(MyApp)
    ],
    bootstrap: [IonicApp],
    entryComponents: [
        MyApp,
        HomePage
    ],
    providers: []
})
export class AppModule {}
```

然后在页面模板中直接这么用就可以了：

```
<my-component></my-component>
```

你后续基本上不需要创建这样的组件，因为Ionic以及提供了大部分你需要用到的组件（列表，标签页，按钮，输入框等等）。如果你需要的组件Ionic没有的话，那么你就可以考虑来看看怎么创建你自己的组件了。
注意：你可以在项目中通过 *ionic g component MyComponent* 命令来生成一个组件。

创建指令

之前也讲过，组件和指令非常像，但是总的来将组件是用来创建一个全新的元素，而指令是用于修改已存在组件的行为的。
自定义指令的类代码是这样的：

```
import { Directive } from '@angular/core';

@Directive({
    selector: '[my-directive]'
})

export class MyDirective {
    constructor() {

    }
}
```

在这条指令中，我们有一个跟组件一样的 *selector* -- 但是又有少许不同。他不是用作标签名，而是作为元素的属性使用的。在Ionic 2中，你会经常用到这个，例如，在按钮上：

```
<button ion-button>
```

或者，列表上：

```
<ion-list no-lines>
```

在这里例子中我们创建的自定义指令可以在任何地方使用，如：

```
<button my-directive>
```

注意，这个指令实际上是没有模板的。尽管我们通常将应用中的任何特性都成为‘组件’，技术上讲组件是有一个类和一个模板（视图）组成 -- 如果他没有视图的话那么他就不是一个组件（他更像一个服务或者提供者）。

这里我普及点基础知识，我觉得需要知道 **ElementRef**。他使我们可以访问到添加了指令的元素上去。你可以在指令中添加如下：

```
import { Directive, ElementRef } from '@angular/core';

@Directive({
    selector: '[my-directive]'
})

export class MyDirective {
    constructor(element: ElementRef) {
        this.element = element;
    }
}
```

跟自定义组件一样他也需要去 **app.module.ts** 文件里面声明。
注意：可以通过命令 *ionic g directive MyDirective* 来生成一个新的 指令。

创建管道（Pipe）

第一眼看管道可能会有点复杂，但是实现真的很简单，他们看起来是这样的：

```
import { Injectable, Pipe } from '@angular/core';

@Pipe({
    name: 'myPipe'
})

@Injectable()
export class MyPipe {
    transform(value, args) {
        //do something to 'value'
        return value;
    }
}
```

注意管道也是一个 **@Injectable**，我们大概的了解一下。他的理念是任何你传入管道的东西都将进入 **tranform** 函数，你可以在其中对值进行任何处理，然后返回新值。然后返回的值将会被渲染到屏幕上，而不是初始值。我们可以在模板中这样使用：

```
<p>{{someValue | myPipe}}</p>
```

不论 **someValue** 是什么他都会在展示之前经过自定义的管道处理。再次声明，使用自定义管道之前一定确保你已经在 **app.module.ts** 中导入和添加。

注意：可以通过命令 *ionic g pipe MyPipe* 来生成新的管道

创建注入（Injectable）

注入允许你创建一个服务在整个应用中使用（就像应用和外部或者网络数据服务之间的接口）。注入也可以作为哦哦'提供者（Provider）'。Ionic CLI自动生成的**@Injectable**看起来是这样的：

```
import { Injectable } from '@angular/core';
import { Http } from '@angular/http';
import 'rxjs/add/operator/map';

@Injectable()
export class Data {
    data: any;

    constructor(public http: Http) {
        console.log('Hello Data Provider');
    }

    load() {
        if (this.data) {
            return Promise.resolve(this.data);
        }
        // don't have the data yet
        return new Promise(resolve => {
            // We're using Angular Http provider to request the data,
            // then on the response it'll map the JSON data to a parsed JS object.
            // Next we process the data and resolve the promise with the new data.
            this.http.get('path/to/data.json')
              .map(res => res.json())
              .subscribe(data => {
                // we've got back the raw data, now generate the core schedule data
                // and save the data for later reference
                this.data = data;
                resolve(this.data);
            });
        });
    }
}
```

以上代码创建了一个名为 **Data** 的提供者用于从JSON数据源（可以是一个本地的JSON文件，也可以是外部的JSON文件或者服务端响应）加载数据。他返回一个promise，promise允许在 **http** 请求执行完成之后获取数据。如果数据已经加载过了，那么他会直接返回数据（通过一个promise返回）。后续会深入了解如何通过 **http** 获取数据，目前我们只需要关注注入的基本知识。

所以，如果我想获取服务返回的数据，我们会在需要用到的类中注入他：

```
import { Component } from '@angular/core';
import { Data } from '../../providers/data';

@Component({
    selector: 'home-page',
    templateUrl: 'home.html'
})

export class MyPage {

    constructor(public dataService: Data){

    }
}
```

同时将他添加到 **app.module.ts**：

```
providers: [Data]
```

然后，你就可以在任何函数中通过 ***this***.*dataService*来使用了，例如：

```
this.dataService.load().then((data) => {
    console.log(data);
});
```

注意，我们这里用到了 *then()*，因为返回的是一个promise，所以我们需要等到promise完成才能访问数据。你可以考虑扩展一下提供一个**save**函数：

```
this.dataService.save(someData);
```

当然，提供者并不是专门用来获取数据的，你也可以用它做别的事情 -- 这个只是个使用比较广泛的用例而已。
注意：可以通过命令 *ionic g provider MyProvider* 来自动生成一个注入。

总结

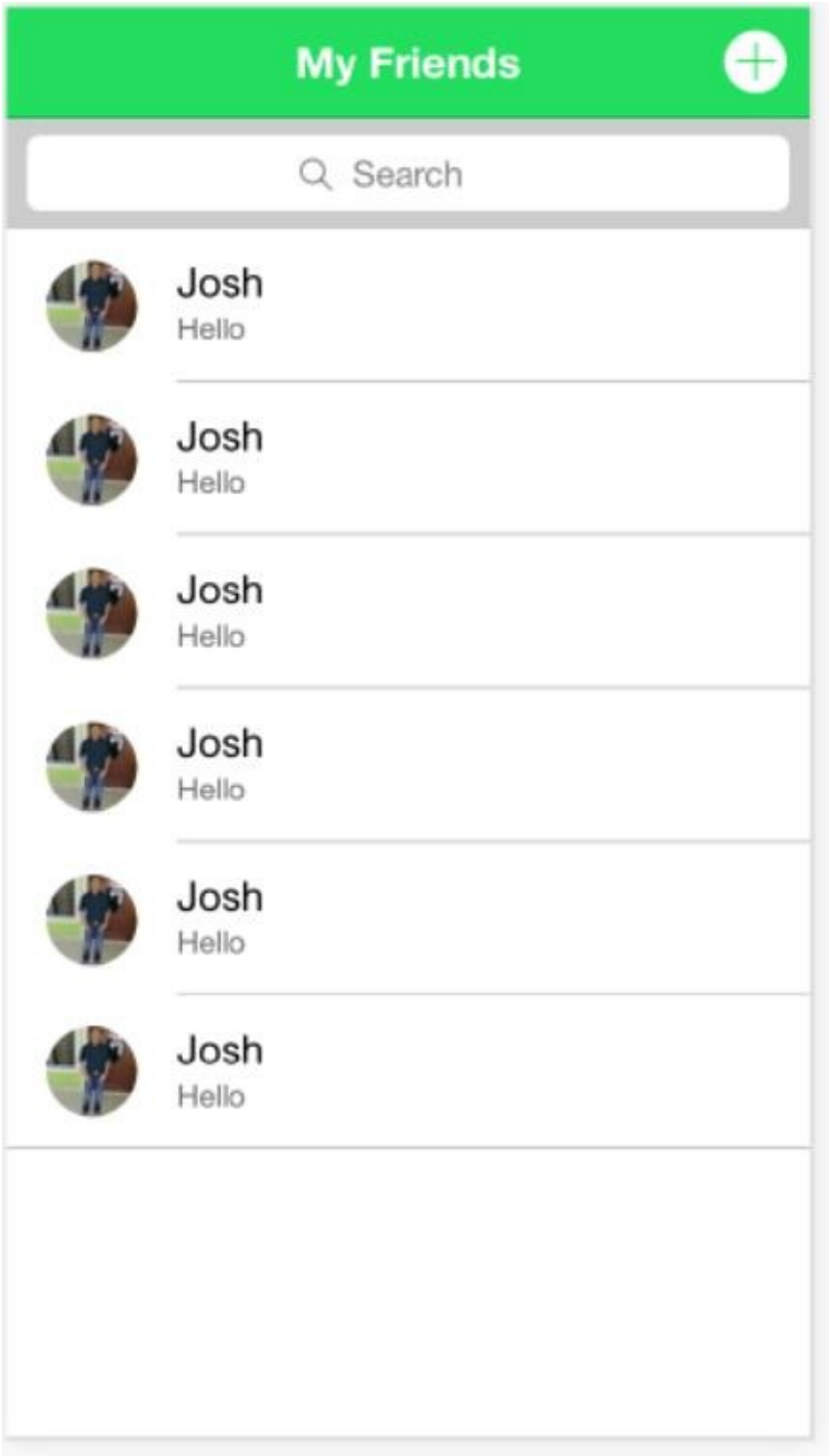
我们广泛详尽的涉及了Ionic 2里面不同类型的类的创建，当然，还有更多需要我们去学习的。但是你现在知道得足够多了，在开始做例子的时候，所有东西看起来也不会那么奇怪与陌生了。

第六课：模块

模板，个人认为是Ionic 2中最有趣的东西。也是框架能力闪耀的地方。看看下面的代码：

```
<ion-header>
    <ion-navbar color="secondary">
        <ion-title>
            My Friends
        </ion-title>
        <ion-buttons end>
        <button ion-button icon-only (click)="doSomethingCool()"><ion-icon name="add-c
ircle"></ion-icon></button>
        </ion-buttons>
    </ion-navbar>
</ion-header>
<ion-content>
    <ion-searchbar (input)="getItems($event)"></ion-searchbar>
    <ion-list>
        <ion-item *ngFor="let item of items">
            <ion-avatar item-left>
            <img [src]="item.picture">
            </ion-avatar>
            <h2>{{item.name}}</h2>
            <p>{{item.description}}</p>
        </ion-item>
    </ion-list>
</ion-content>
```

上面代码在没有额外样式的情况下，看起来将会是这样的：



看起来没那么惊艳，但是我们就已经用简单的几行代码设置好了一个复杂的布局，加点自定义样式的话，我们就可以得到一个非常时尚的界面了。稍后我们会从各个方面彻底的了解在Ionic 2中创建模板，但是目前我需要给你找找一个完整的页面模板看起来是什么感觉，以及使用Ionic提供的组件是多么的简单。

Ionic 2中的模板语法有很多知识需要学习，如果你之前有使用过Ionic 1的话，你会发现他有一些重大的变更 -- 所以我们需要具体的学习一下模板。

此后也会涉及到其他的一些知识，但是这个是我认为这开始使用Ionic 2之前最后学要学习的'核心'知识 -- 一旦你拿下了类和模板，那么你就可以直接可以开始制作一些东西。所以，我们先转入一些基础理论知识的学习，然后在做学一些练习示例吧。

* 语法

*可能是Ionic 2里面最让人困惑的语法之一了。你会经常遇到一些这样的代码：

```
<ion-item *ngFor="let item of items">
```

或者

```
<p *ngIf="someBoolean"><p>
```

诸如此类。在Angular 2中 语法用与创建一个嵌入模板的快捷方式，所以，当我们使用 ngIf的时候，上面的代码展开来就是：

```
<template [ngIf]="someBoolean">
    <p></p>
</template>
```

使用模板的原因是Angular 2将模板看作是DOM块，这样一来就可以动态操作他了。所以，在上面的 *ngIf 范例中我们不会只按照字面意思将他渲染到DOM，

```
<p *ngIf="someBoolean"></p>
```

如果 *someBoolean* 等于true的话，会显示：

```
<p></p>
```

false的话就不显示。同理，当我们使用 *ngFor的时候，我们不只是按照字面上去渲染：

```
<p *ngFor="let item of items">{{item.name}}</p>
```

我们会针对每个条目分别以段落的方式渲染出来：

```
<p>Bananas</p>
<p>More Bananas</p>
<p>Pancakes</p>
```

要使用这个功能的话，我们需要使用，但是手动写这些模板又很繁琐，所以 语法是用来简化这些繁琐的操作的。
解释得这么清楚了，我们来具体的看看想 ngIf 和 *ngFor 这样的指令的使用方法。

循环

很多时候你会循环大量的数据 -- 例如当你有一个文章列表你想要将所有文章的标题渲染到一个列表。我们就可以用Angular 2 提供的**ngFor**指令来完成这个任务了 -- 看起来大概是这样的：

```
<ion-list>
    <ion-item *ngFor="let article of articles" (click)="viewArticle(article)">
        {{article.title}}
    </ion-item>
</ion-list>
```

上面的例子中，我们创建了一个，然后对于**articles**数组里面的每个**article**我们都添加了一个。在之前的基础部分我讲过了使用let来创建一个局部变量，我们这里就用到了。他允许我们访问我们当前循环到的**article**，我们使用这个变量来获取他的标题然后在列表上渲染出来，同时，在用户点击的时候将他传入到*viewArticle*函数中。
通过将当前article的引用传入到*viewArticle*函数，我们可以用来做类似弹出文章新页面的操作。

条件

有时候你想着符合特定条件的情况下展示模板特定的部分，可以使用一些方法来做到：

```
<div *ngIf="someBoolean">
```

当*ngIf*的表达式值为true的时候，那么他附加到的节点才会被渲染出来。在本例中，只有在*someBoolean*为true的时候，才会被添加到DOM，反之则不会。
*ngIf*在布尔场景（true和false）下非常厉害，但是有时候你需要根据大量不同的值来处理。这种情况下就得使用*ngSwitch*：

```
<div [ngSwitch]="paragraphNumber">
    <p *ngSwitchCase="1">Paragraph 1</p>
    <p *ngSwitchCase="2">Paragraph 2</p>
    <p *ngSwitchCase="3">Paragraph 3</p>
    <p *ngSwitchDefault>Paragraph</p>
</div>
```

在这个例子中，我们使用*ngSwitch*来检查*paragraphNumber*的值。哪个*ngSwitchCase*语句匹配到了这个值，就会是以哪个作为DOM元素去渲染，如果没有匹配上的值，那么就用*ngSwitchDefault*元素。
还可以通过*hidden*属性去根据条件去显示或者隐藏一个元素。
例如：

```
<ion-avatar [hidden]="hideAvatar" item-left>
```

在这个例子中，当*hideAvatar*为**true**的时候，这个元素将会隐藏，当为**false**的时候则显示。使用这个方法的时候，你的类定义里面应该有**this**.*hideAvatar*变量存在，你可以通过给这个变量赋值来控制元素的显示和隐藏。
不但可以根据条件显示整个元素，还可以根据条件给元素添加不同的类，例如：

```
<ion-avatar [class.my-class]="showMyClass" item-left>
```

这个跟上面的*[hidden]*方法类似，但是他不是根据条件显示和隐藏元素，他根据条件来添加CSS里面定义好的类。这个方法非常实用，例如，当你想要用来区分列表里面已读和未读信息的时候。

Ionic 2 模板组件

目前为止谈到的基本上都是Angular 2的东西，没有Ionic特有的（除了模板里用的和之外）。这些语法和一些Ionic特有组件将在你的模板里贯穿始终。我们现在要学习一些Ionic特有的东西了，先从Ionic 2页面模板的基本布局开始：

```
<ion-header>
    <ion-navbar>
        <ion-title>
            Home
        </ion-title>
    </ion-navbar>
</ion-header>

<ion-content class="home">
    <ion-card>
        <ion-card-header>
            Card Header
        </ion-card-header>
        <ion-card-content>
            Hello World
        </ion-card-content>
    </ion-card>
</ion-content>
```

这个是你使用blank布局自动生成的模板代码。这里有两个很重要的组件，基本每个模板都会出现的：和。
元素只是简单的用来包含页面的主要内容（本案例中的‘卡片’），并允许滚动。注意，他有个名为‘home’的类，如果你查看**home.scss**文件的时候，你会发现里面有‘home’的类定义。他没做什么特别的事情，他只是一个约定，允许你对的样式单独进行变更（记住，即使你只是在**home.scss**里添加了样式，这样新样式还是会应用到整个项目的，文件分离只是为了架构而已）。
两者之间更为有趣的是。这个是用来添加页首的，里面可以添加页标题，以及左右按钮。虽然这可以不大符合审美，他还有很多内置的导航智能。如果你是压入 **push**一个新页面（后面会涉及），那么里面会自动出现一个返回按钮允许用户返回之前页面，而不用你手动去添加。
上面部分包含的基本模板语法在Ionic 2页面中会经常见到，其他需要做的就是拖入和配置Ionic 2提供的大量组件（如果你喜欢冒险，那么自建组件）。
现在我们看一下如何在模板中实现一些Ionic组件。我们不会全部组件都讲到，因为它们是在是太多了，我们只是来尝尝鲜。完整的组件列表，可以在Ionic 2文档中查阅。

列表 List

列表上移动应用中使用最广泛的组件之一，他们提佛那个一个很有趣的挑战。在本地应用上滑动那种丝滑的感觉，那种顺滑的加速和减速，感觉起来真的是爽呆了 -- 这种感觉很难复制。幸运的是，你不用担心这个，Ionic 2为了解决了所有的难点，下面这样就可以简单的使用列表了：

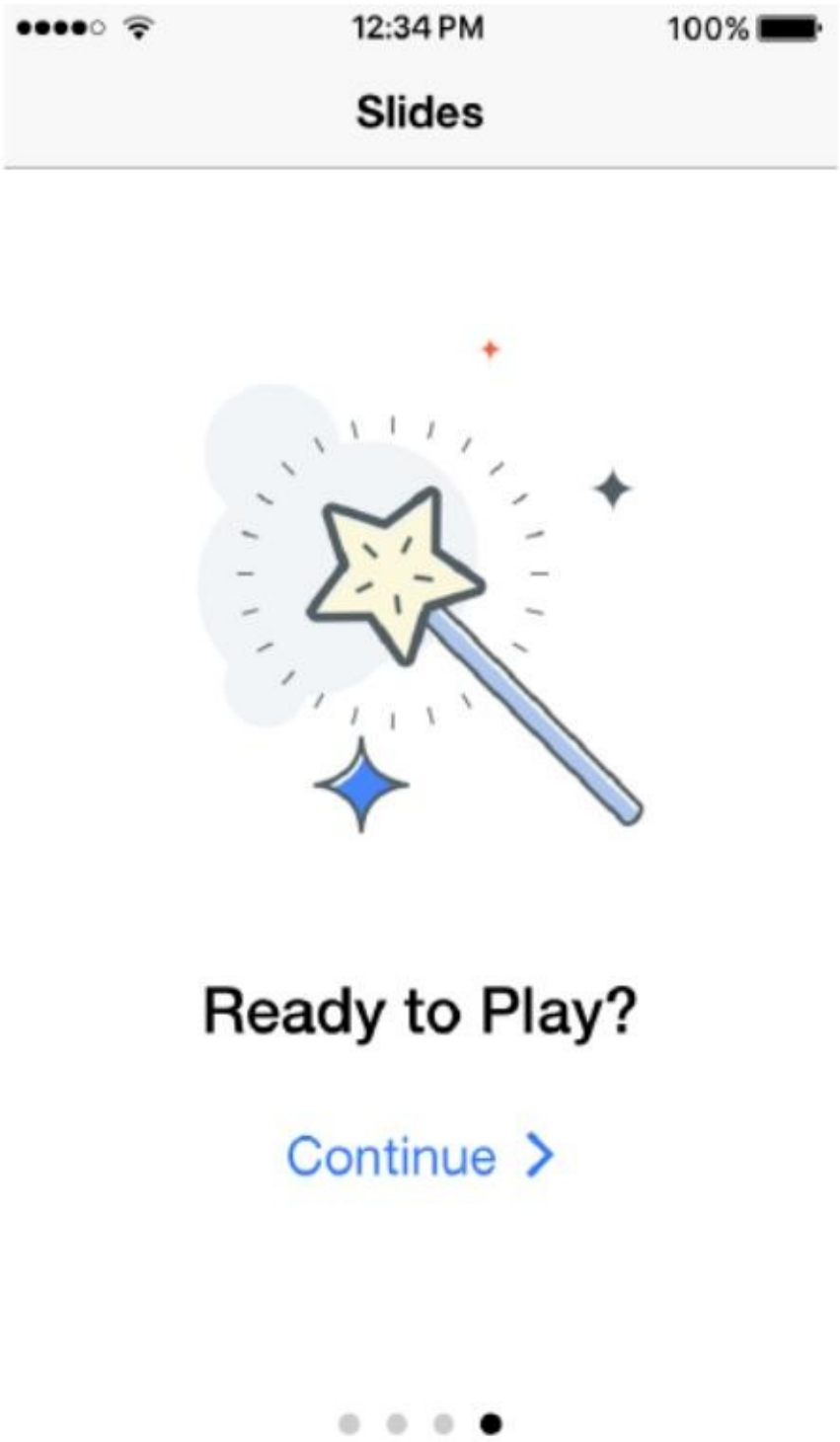
```
<ion-list>
    <ion-item>Item 1</ion-item>
    <ion-item>Item 2</ion-item>
    <ion-item>Item 3</ion-item>
</ion-list>
```

或者如果你想要根据类定义里面的一系列数据来动态窗台你的列表：

```
<ion-list>
    <ion-item *ngFor="let item of items" (click)="itemSelected(item)">
        {{item.title}}
    </ion-item>
</ion-list>
```

滑块 Slide

滑块是另一个移动应用的通用组件之一，滑块看起来是这样的：



在你有大量的图片或者页面想要通过用户左右滑动来展示的情况下，滑块就可以上场了。与列表一样，使用滑块也非常简单：

```
<ion-slides [options]="slideOptions">
    <ion-slide>
        <h2>Slide 1</h2>
    </ion-slide>

    <ion-slide>
        <h2>Slide 2</h2>
    </ion-slide>

    <ion-slide>
        <h2>Slide 3</h2>
    </ion-slide>
</ion-slides>
```

这里用到了一个\容器，然后每个单独的滑块都分别使用\来定义。也可以提供一些选项来定义滑块的行为；例如是否循环是否分页。（后续会有完整示例）

输入 Input

Ionic 2中使用\来代码\。跟普通的\一样，可以根据你想要获得的信息来给他指定类型，使用Ionic版的输入框可以享受Ionic为移动做的自定义设计的好处。

```
<ion-list>

    <ion-item>
        <ion-label fixed>Username</ion-label>
        <ion-input type="text" value=""></ion-input>
    </ion-item>

    <ion-item>
        <ion-label fixed>Password</ion-label>
        <ion-input type="password"></ion-input>
    </ion-item>

</ion-list>
```

就跟定制的\一样，Ionic提供了其他输入自定义的输入组件，如\，\，\以及\。

格子 Grid

Grid组件非常强大，可以用来创建复杂的布局。如果你对CSS框架比较熟悉，例如Bootstrap，那么你应该很熟悉这个概念。当往你的模板中添加组件的时候，通常都是一个接一个的现实，但是有了Grid你可以实现任何你想要的布局。
他的工作方式是将元素以行与列（放在行里面）的方式边靠边。例如：

```
<ion-row>
    <ion-col></ion-col>
    <ion-col></ion-col>
</ion-row>
<ion-row>
    <ion-col></ion-col>
    <ion-col></ion-col>
    <ion-col></ion-col>
</ion-row>
```

以上代码将创建一个两行的布局，上面的布局有两列，下面的布局有三列。默认所有元素均匀分布，你也可以指定列的宽度：

```
<ion-row>
    <ion-col width-10></ion-col>
    <ion-col width-20></ion-col>
    <ion-col width-25></ion-col>
    <ion-col width-25></ion-col>
    <ion-col width-20></ion-col>
</ion-row>
```

以上代码将创建单独的一行，里面有5个不同宽度的列（你可以将列宽设置为100！）。
所有可用宽度请参考文档。

图标 Icons

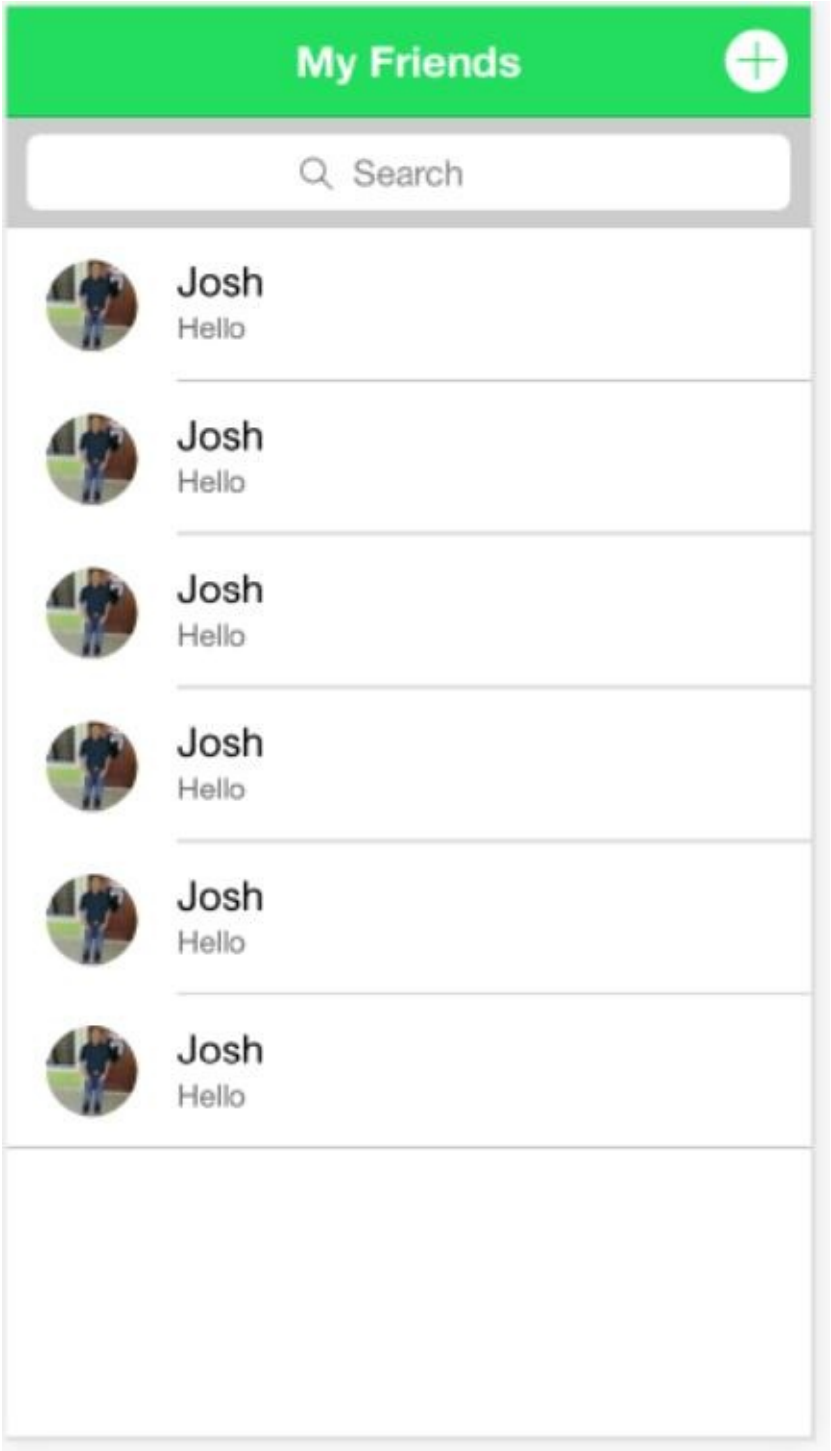
现在的应用大量用到图标，他比文本的伟大之处在于他可以漂亮的表达事物的表达内容。大部分时候，他更易用比按钮加上一个‘添加条目’之类的标签更好看。
Ionic提供超多的图标，例如：

```
<ion-icon name="heart"></ion-icon>
```

你只需要指出你需要使用的图标的名字即可。甚至他可以根据iOS和Android平台来切换不同的图标以更好的适配平台。所有可用图标可在此处查看。
Ionic还有很多其他的组件，即使这里已经讲过的组件，也有很多其他的细节需要你去了解，所以建议先阅读一下文档熟悉一下。

第七课 样式与主题

我很喜欢Ionic的原因是，默认的组件都是开箱即用型的。一切看起来很整齐，平滑，整洁，同时也有点枯燥。我喜欢简单，平平淡淡才是真，但是，也许你不想你的应用看起来跟其他应用没什么两样。举一下我们在模板中讲过的例子：



界面看起来简单明了，但是明显没有赢得任何设计大奖的可能性。他使用的都是默认的样式，没有半点自定义的东西。如果你看一下我本课程中制作的应用的时候，会发现他们基本上都有自定义样式：

一些应用只有简单的自定义样式，而简单的自定义样式就可以让它耳目一新。一些应用有更复杂的自定义样式，他完全使项目焕然一新。

我当然不是设计大神，但是我觉得我给这些应用创建的主题让他们看起来更舒服也给了应用一些特色。本课程中将给你展示使用不同方法自定义你的Ionic 2应用， 以及主题背后的原理。

介绍Ionic 2主题

给Ionic 2应用定制样式在本质上和给网站定制样式没什么区别。我经常看到这样的问题：

> 我可以在Ionic里面制作[插入元素/界面]吗？

答案是**yes**。你可以在普通网页上做吗？如果可以的话，那么在Ionic里面也可以。
大部分人都是通过修改CSS文件来改变样式，但是Ionic稍微复杂些，因为Ionic使用的是SASS。同样，SASS不是Ionic或者移动网页应用开发特有的 -- 在很多普通页面上也可以使用 -- 但是很多人对SASS不熟悉所以他们选择使用老旧的CSS。
如果你还不熟悉，**.scss**是SASS的文件类型或者**Syntactically Awesome Style Sheets**牛逼闪闪语法样式表。如果你是第一次看到这个，建议你阅读文档熟悉一下SASS是什么以及他做什么。对于那些着急上手的人们，你在**.scss**里面放的和你在**.css**放的是一样的东西，你可以在里面做很多很酷的东西，例如定义变量然后在不同地方使用。**.scss**文件然后会编译成**.css**文件（和我们在Ionic 2的理念是一样的，我们用有趣的ES6功能去编写代码，然后转译成浏览器实际支持的ES5代码）。
当给你的应用定制主题的时候，主要是去编辑你的 **.html**模板文件 和 **.scss**样式表 -- 绝对不要直接去编辑 **.css** 文件。 *.css* 文件是从*.scss*文件生成的，所以你对 *.css* 文件做的任何更改都将被覆盖。
当你查看创建项目的时候自动生成的文件是，你可以在**theme**文件夹里面看到一些**.scss**文件，我们现在快速了解一下他们是干嘛用的。

- **src/theme/app.scss**用于声明在应用中全局使用的样式
- **theme/variables.scss** 用于修改应用共享变量。在这里你可以编辑一些默认值，例如用于设置应用默认颜色的 *$color*，还有*$list-background-color*和*$checkbox-ios-background-color-on*等等。这个的目的是Ionic使用此处定义的变量来决定许多组件的样式，所以此处是快速变更的绝佳之选。所有此处可以覆盖的变量可以查阅本页。

在你的**theme**文件夹里的这些**.scss**文件之上，你每个自定义组件都有一个对应的。刷新一下你的记忆，Ionic 2中大部分你创建的组件看起来是这样的：
my-component

- my-component.ts
- my-component.html
- my-component.scss

*.ts*里面是类定义，*.html*是模板文件,.*scss*就是组件的样式文件了。虽然不是严格要求的，但是你还是要创建 *.scss* 文件，而不是通过 **app.core.scss** 文件来定义他的样式。如果你是使用Ionic CLI命令来生成组件的时候，*.scss* 文件会自动生成。
为啥？虽然你可以将所有样式放到**app.core.scss**文件里面，但是像上面这样单独创建一个样式表可以带来两个好处：

- 架构 -- 将代码分离出来可以让你的文件保持较小的体积，功能上是一样的，。由于每个

组件的样式都有单独定义的 *.scss*，这样一样就不用费心费力的去找他的定义了。
- 模块性 -- Angular 2和Ionic 2使用这样的组件样式结构上因为模块性。之前的版本，代码互相交织难以分离和重用。现在，特殊组件需要的特殊功能都包含在他自己的文件夹里面，这样一来他就可以很轻松的重用以及拖入到其他项目。

原理我们已经熟悉了，现在我们来开始学习给Ionic 2应用定制样式。

Ionic 2应用定制主题的方法

以下我将讲到一些不同的定制主题的方式，你可以任意选择。有可能什么方式做什么事情有点不大清晰，因为在很多案例中你可以通过不同的方式来达到相同的目的。一般而言，你应该尽量不要通过创建自定义样式的方式来达到你的目的（这个最后讲）。你应该先通过预定义属性或者覆盖SASS变量来达到你的目的。如果这样无法做到的话，你在考虑创建你的自定义样式。不要担心，尽量使事情简单化。

1.属性

改变样式最简单的方式是给元素添加你正在使用的属性。如上所述，SASS用来定义一些颜色：

- primary
- secondary
- danger
- light
- dark
- favorite

你可以在**app.variables.scss**文件中看到他们的定义：

```
$colors: (
    primary: #387ef5,
    secondary: #32db64,
    danger: #f53d3d,
    light: #f4f4f4,
    dark: #222,
    favorite: #69BB7B
);
```

可以看到Ionic提供的这些默认的颜色是什么，你也可以覆盖以上颜色为任意你喜欢的色值。大部分你添加**primary**属性的元素将会变成蓝色，添加**danger**属性的会变成红色。如果你更改他们的话**primary**可以显示为紫色，**danger**可以将元素变成粉色。
如果想要对一个按钮使用**secondary**的话，我们应当这么做：

```
<button color="secondary"></button>
```

但是如果我想对一个导航条使用*secondary*颜色的话：

```
<ion-navbar color="secondary"></ion-navbar>
```

记住，这些属性不只是改变每个元素的颜色这么简单，一些属性还可以改变元素的位置：

```
<ion-navbar color="secondary">
    <ion-buttons end>
        <button ion-button color="primary">I'm a primary coloured button in the end po
sition of the nav bar</button>
    </ion-buttons>
</ion-navbar>
```

上面的代码中用到了*end*属性来决定按钮在哪里显示。同时，我们对*button*使用了*ion-button*属性，这样Ionic知道对这个按钮应用样式。我们也可以控制一个列表是否可以有边缘：

```
<ion-list no-lines></ion-list>
```

甚至用来控制列表条目是否显示箭头来说明是是否被点击：

```
<ion-item detail-none></ion-item>
```

还有大量的其他这样的属性，所以在使用Ionic制作组件之前一定要多看看文档。*no-lines*属性对于从列表中移除线条真的非常简单，但是如果你不知道这个属性的存在的话（可能性还蛮大的），那么你就不能继续创建你的自定义样式了。这就是我为什么建议你优先使用属性，因为可以帮你节省大量的工作量。

2.SASS变量

接下来可以用来控制应用样式的方法是更改默认的SASS变量（例如编辑我们上面提到过的*$color*）。这个方法非常方便，因为他可以让你大范围的对指定的事物进行变更。我之前也提及SASS变量，但是基本上在你的*.scss*中你可以这样：

```
$my-variable: red;
```

然后在所有的*.scss*文件中你可以用*$my-variable*。这样一来，假设你想将20个不同元素的背景设置为红色，与对每个进行如下变更：

```
background-color: red;
```

对比，你可以这样去操作：

```
background-color: $my-variable;
```

这样做的好处是当你想把背景色从红色环岛绿色的时候，你只需要把这一个变量设为绿色即可-- 用不着去一个一个的编辑。这就是为啥你发现变量的都是以**primary**和**danger**的方式去命名，而不是具体的**blue**和**red**。有时候当你想将你的主色变为紫色，但是如果你给一个名为*$my-blue-color*设为紫色的话，会让你的代码看起来很混乱。
可能你需要自己定义的变量不多，但是Ionic定义和使用了大量的变量，你可以很轻松的将他们覆盖为其他东西。我们看一下其中一些变量：

- $background-color
- $link-color
- $list-background-color
- $list-border-color
- $menu-width
- $segment-button-ios-activated-transition

可以查看文档来具体了解这些变量以及他的默认值，虽然他们的名字很清晰的表达出来。就像你在上面的范例中看到的一样，他们简直不能更清楚了。
对这些变量进行编辑很简单，打开**app.variable.scss**然后插入你自己的定义就可以了。以下是本书一个例子的**app.variable.scss**：

```
$colors: (
    primary: #387ef5,
    secondary: #32db64,
    danger: #f53d3d,
    light: #f4f4f4,
    dark: #222,
    favorite: #69BB7B
);

$list-background-color: #fff;
$list-ios-activated-background-color: #3aff74;
$list-md-activated-background-color: #3aff74;

$checkbox-ios-background-color-on: #32db64;
$checkbox-ios-icon-border-color-on: #fff;

$checkbox-md-icon-background-color-on: #32db64;
$checkbox-md-icon-background-color-off: #fff;
$checkbox-md-icon-border-color-off: #cecece;
$checkbox-md-icon-border-color-on: #32db64;
```

以上范例中，对一些默认的颜色进行了改动，iOS和Android的一些特有样式也有覆盖。
注意这里有用到 **md** ，这是 material design的简写，是用于Android的。Ionic 2根据平台约定无缝调节样式显示 -- 对于Android来将意味这使用了 material design。
译者： Material Design
编辑这些默认的SASS的伟大之处在于，一处修改，应用内用到之处都生效。那些使用别的变量的值的变量，如果你想在CSS中手动去修改的话，工作量可大了。

3.配置

另一个改变样式的快捷方式是修改**app.module.ts**里面提供给**IonicModule**的**Config**对象。
大体上这里是用于设置应用内广泛的默认行为，例如按钮和页签的摆放，图标的使用样式，过度动画等等。通常是最好不要对他进行修改，除非你有特别的需要在这里修改的理由，因为Ionic会根据平台自动调节显示 -- 跟config过不去就是跟Ionic过不去。
有时候你想强制事物一个指定行为，**Config**就是用来做这样的特例的。一下范例是用来这么做的：

```
IonicModule.forRoot(MyApp, {
    backButtonText: 'Go Back',
    iconMode: 'ios',
    modalEnter: 'modal-slide-in',
    modalLeave: 'modal-slide-out',
    tabbarPlacement: 'bottom',
    pageTransition: 'ios'
})
```

如果你想强制iOS使用Material Design你可以这样去设置Config：

```
IonicModule.forRoot(MyApp, {
    mode: 'md'
})
```

再次强调，除非有特别的理由，否则不要这样做。也许你喜欢material design，或者你习惯了material design，但是你的iOS用户看到的东西跟你的不一样。记住这一点只偶，Config也允许你为特殊平台做特殊配置：

```
IonicModule.forRoot(MyApp, {
    tabbarPlacement: 'bottom',
    platforms: {
        ios: {
            tabbarPlacement: 'top',
        }
    }
})
```

关于Config对象的更多信息，请查阅文档。

4.自定义样式

在我们聊到使用属性改变元素颜色之前，假设你可以将这些属性覆盖为你喜欢的任意值，设置primary，secondary，danger变量来匹配你的设计样稿的，然后使用它们来设置你的元素，而不是定义你的自定义CSS类，这也不是为一个好的途径。
但是有些时候，你想要定义一些简单的老CSS类来达到你的需求。如果变量是针对整个应用的话，那么可以将他定义在**global.scss**中，如果是单独针对某个组件的，那么可以定义在此组件的*.scss*中。
当然，你也可以直接在元素上通过style标签来自定义样式，但请谨慎使用。

如上，你可以通过一些不同的方法修改你的Ionic 2应用的样式。总的来讲，尽量去简单的达到你的需要。尽量使用属性和SASS变量，因为他使你的生活如此简单。

之前也说过，Ionic在iOS和Android之间平滑调节UI，所以当你越'骇客式'或者'暴力'的去自定义样式，你就越粗鲁的破坏力这个行为。

第八课：导航

如果你有Ionic 1或者Angular 1的背景，那么你以前应该处理过URL，状态等之间的路由导航。Ionic在这个上的专注点是使用一个导航栈，他引入了**pushing** 压入视图到**navigation stack**导航栈和**popping** 弹出。在具体了解Ionic 2中是如何具体实现这个之前，我们先来了解一下这个概念。

压入与弹出

想象一下你的**root page** 根页面是一张画有小猫的纸，然后你将这张纸放到桌子上。那么现在他是桌子上唯一的一张纸，你现在正在从上往下盯着他。由于他目前是桌子上唯一的一张纸，所以你可以看到这个小猫图片：

现在，我们假设你想看另一张不同的纸（也就是去到另一个页面），那么你会将另一张纸压入到你的纸张栈上面。我们就假设这是一张小狗的图片吧，你拿出另一张纸，然后将他放到小猫上面：

小猫还在那里，但是我们已经看不到他了，因为他在小狗后面。我们更深入一点，假设我们压入一张小牛的图片，看起来应该是这样的：

小猫和小狗都在那里，但是小牛在最上面所以我们只看得到小牛。现在我们稍微倒过来一点。由于纸张都是按顺序的堆起来的，所以我们可以非常容易的通过弹出来回溯他们。如果你想回到小狗的图片，你可以弹出纸张栈，移除当前顶部的纸张（小牛）。如果你想回到小猫的图片，你可以再次弹出纸张栈，也就是移除当前顶部的小狗图片。现在我们回到了开始

的地方。
你应该看到了这种导航方式对维护历史记录多方便，在导航到子视图的时候非常好理解，但是讲压入和弹出就没那么好理解了。有时候你想直接回到某个页面而不触发这个变更。（例如，进入应用之前的登录界面，甚至是通过菜单改变的小部分区域）
在本例中，我们更改根页面，以我们桌子上的图片打比方，即忽略当前栈中的其他图片只关心桌子上当前显示的图片：

上面的例子中，我看到一个小牛的页面作为根页面，与将他显示到栈顶来比，它只有它自己。
刚开始，将页面设置为根页面来导航还是将他压入栈来进行导航很难理解。一般来讲，如果你想要转到的页面上当前页面的子页面，或者如果你想使用导航的回退功能，那么用压入的方式。举例，当我正在浏览艺术家列表，然后点击其中一个以显示详情就需要将他压入。当我浏览了大量页面表单然后点击'下一个'进入到表单的第二页的时候，我就需要压入那个第二页了。
如果你想要转到的页面不是当前视图的子页面，或者他是应用的不同部分，这个时候你就可以使用设置根页面的方式了。例如，当你有一个在进入应用之前的登录也，在用户登录验证成功之后，你就需要该变更应用的根目录里。如果你有一个侧边菜单
有**Dashboard**，**Shop**，**About**和**Contact**选项，在用户选择这些选项的时候你就可以设置相应的根页面。
始终记住根页面和根组件不一样，普通来讲，根组件（在**app.module.ts**里定义的）定义了根页面是什么 -- 根页面在应用中到处都可以改，然后跟组件不能更改。

Ionic 2中的基本导航

好了，我们了解了原理，所以现在我们来深入一个实际的Ionic 2范例看一下压入，弹出以及设置根页面以及如何在页面之间传递参数。
所有这些的核心是Ionic提供的**NavController**。你会经常看到Ionic 2应用中导入它：

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

@Component({
    selector: 'home-page',
    templateUrl: 'home.html'
})
export class HomePage {

    constructor(public nav: NavController) {

    }
}
```

我们注入了**NavController**然后创建了他的应用，这样我们在类里面可以使用它。也许你猜到了，NavController帮助我们控制导航 -- 所以我们来看一下如何使用他来压入和弹出。
压入页面需要一个页面，将他放到导航栈的顶部（这样将会把它设置为当前页面），大概是这样的：

```
this.nav.push(SecondPage);
```

这里用到了我们之前创建的**NavController**引用，你只需要提供需要导航的页面的引用就可以了，当然这个页面也需要在页头导入：

```
import {SecondPage} from '../second-page/second-page';
```

同时添加到**app.module.ts**的*entryComponents*和*declarations*数组里。当触发压入代码的时候，应用将会转到新的页面。当你压入一个页面的时候，导航栏（假定你有）会自动出现一个‘返回’按钮，所以通常你是不同担心通过弹出导航回之前的页面，因为‘返回’按钮自动帮你完成了这个操作。
有些情况下你需要手动弹出一个页面，你可以这样去做：

```
this.nav.pop();
```

非常简单，对不对？我之前讲过，还有其他方法来切换页面，也就是设置根页面。如果你看一眼**app.ts**的时候你发现下面这行代码：

```
rootPage: any = MyPage;
```

在根元件里面声明*rootPage*会设置根页面，因为根组件的模板是这样的：

```
<ion-nav [root]="rootPage"></ion-nav>
```

这样我们将会设置的root属性为*rootPage*定义的值。当你在应用的任何角落想要更改根页面的时候，可以通过我们的老朋友**NavController**来实现 -- 你只需要调用他的*setRoot*方法：

```
this.nav.setRoot(SecondPage);
```

页面之前端的参数

移动应用的一个基本需求是指页面之间传递参数。一个很常见的例子是当使用'Master Detail'模式（也就是你有一个列表的条目，在你点击其中一个条目的时候跳转到另一个页面显示选中条目详情）的时候。当你导航到详情页面的时候，我们需要知道我们要展示的条目的数据，这些数据需要从之前的页面传过来。在Ionic 2中，可以通过**NavParams**来实现。首先，你需要在压入（setRoot里面也可以用）调用的时候传入需要传递的数据：

```
this.nav.push(SecondPage, {
    thing1: data1,
    thing2: data2
});
```

这跟我们之前做的是一模一样的，除了增加了一个额外的参数，这个参数是一个包含了我们需要传递到**SecondPage**的数据的对象。然后，在接受页面，我们导入**NavParams**然后将它们注入到构造器：

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

@Component({
    selector: 'second-page',
    templateUrl: 'second-page.html'
})

export class SecondPage {
    constructor(nav: NavController, navParams: NavParams){

    }
}
```

然后你就可以通过以下方式来获取传入的参数：

```
this.navParams.get('thing1');
```

导航组件

Ionic提供的一些组件都有不一样的导航行为。这些不是核心导航概念，但是在应用中它会和导航相冲突。所以，我们来看一下他们是什么和何时使用它们。

模态框 **Modals**

也许你已经熟悉了模态框的理念。在网络开发中，模态框基本上是一些将所有其他内容挡住后面的弹出框。通常模态框有'lightbox'样式，一个暗化的背景和聚焦的内容区域。
Ionic里面的模态框也是一样的，他在你的内容之上弹出，但是他看起来跟其他正常页面没有区别。通常来讲，当你想给用户一个启动然后直接移除而不是回退的视图的时候，你会用到模态框，而不是压入页面。
模态框有一个很酷的能力就是他支持者移除的时候向启动他的页面传回数据。例如，你可以创建一个这样的模态框（记得导入和注入**ModalController**）：

```
let myModal = modalCtrl.create(MyPage);
myModal.present();
```

注意，模态框是'呈现'的，而不是压入到导航栈。当我们想要从模态框传回一些数据的时候，我们只要在他呈现之前给他添加*onDismiss*处理器就可以了：

```
let myModal = modalCtrl.create(MyPage);

myModal.onDismiss(data => {
    console.log(data);
});

myModal.present();
```

这样，当模态框移除的时候，我们可以使用他回传的数据做些事情了。移除模态框的方法是在模态框内部调用以下方法：

```
this.view.dismiss();
```

此处的*view*是**ViewController**的引用，他和**NavController**差不多也需要导入和注入到构造器。如果我们想在隐藏的时候往*onDismiss*里面传递数据的话，我们需要这样去操作：

```
let data = {
    thing1: "value1",
    thing2: "value2"
};

this.view.dismiss(data);
```

现在，*data*对象就被传到到启动模态框的页面里面的*onDismiss*处理器里面了。

标签页

标签页是一个非常流行的组件，对我们应用里的导航有很大的冲击。标签页的使用非常简单，在模板里面这么用就可以了：

```
<ion-tabs>
    <ion-tab [root]="tab1Root" tabTitle="Tab 1" tabIcon="navigate"></ion-tab>
    <ion-tab [root]="tab2Root" tabTitle="Tab 2" tabIcon="person"></ion-tab>
    <ion-tab [root]="tab3Root" tabTitle="Tab 3" tabIcon="bookmarks"></ion-tab>
</ion-tabs>
```

然后在类定义中定义这样去定义页面：

```
tab1Root: any = TabOne;
tab2Root: any = TabTwo;
tab3Root: any = TabThree;

constructor(){

}
```

需要记住的是每个标签页有他自己的根页面。你可以这个去想，切换标签页相当于切换不同的根页面，然后在每个标签页里压入和弹出页面。在标签页布局中，可以在每个页签之前切换，每个页签维护了他自己的历史记录。

侧边菜单

侧边菜单实际上做的跟导航没有什么不同，他就是一个UI元素而已，但是他很方便，普通，可以在其中放一些按钮然后通过按钮导航到其他页面（切换页面实际上是手动通过**setRoot**和**push**实现的）。在应用中添加侧边菜单非常简单，这样像下面这样修改你的根组件的模板就可以了：

```
<ion-menu [content]="content">
    <ion-content>
        <ion-list>
            <button ion-item (click)="openPage(homePage)">
            Home
            </button>
        </ion-list>
    </ion-content>
</ion-menu>

<ion-nav id="nav" #content [root]="rootPage"></ion-nav>
```

我们使用来创建一个菜单，同时也需要告诉它我们会附加什么上去。这个就是为什么我们将*[content]*属性设置为一个布局变量*content*，然后在中添加*#content*来定义*content*。基本上这就是说是我们的主要内容区域，我们想让内容在其中显示。

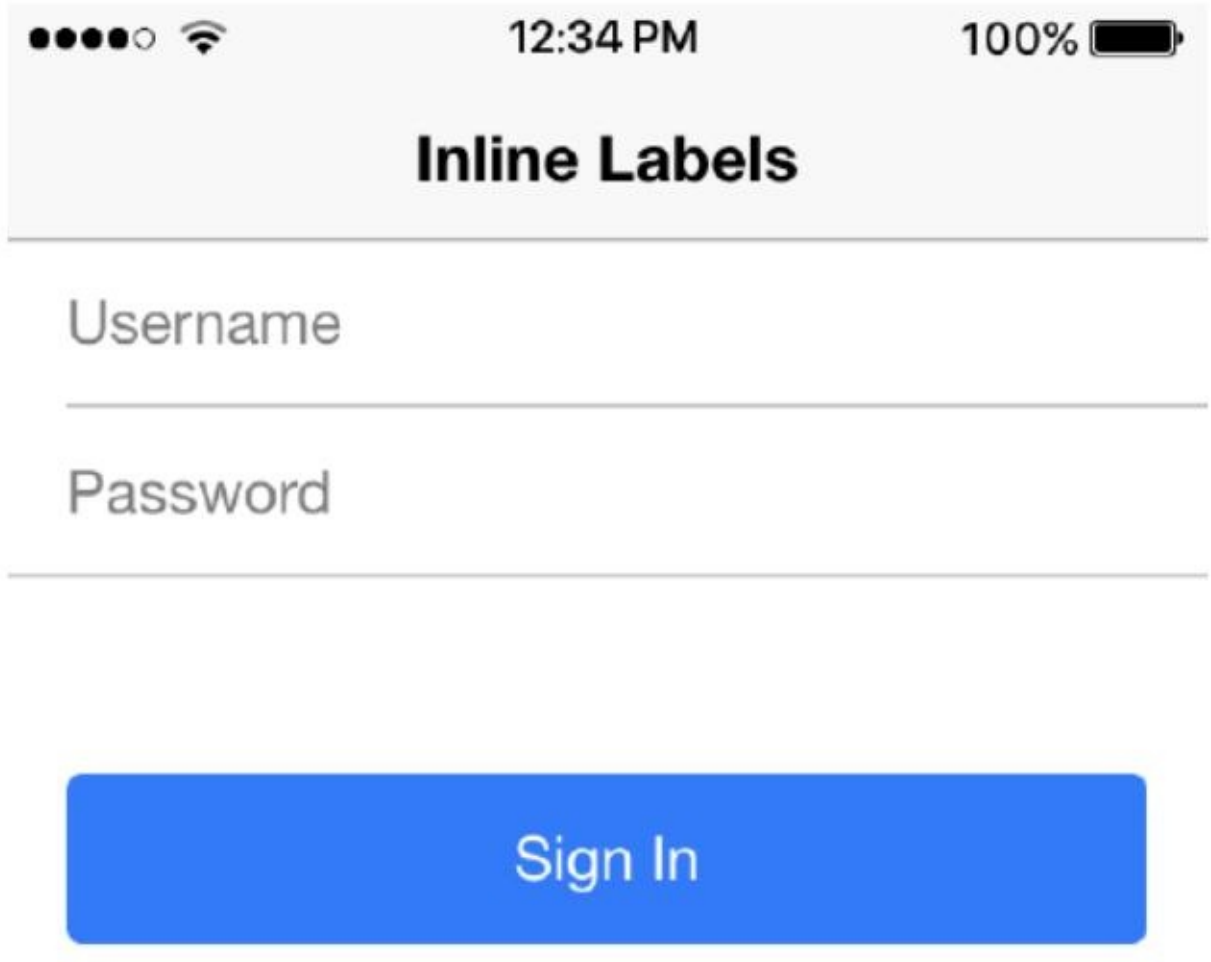
关于Ionic 2的导航多少其他的需要去学习的，但是只要你学好了本课程的这些基本知识，那么在大部分情况下你都不会有什么问题。

第九课：用户输入

不是所有的移动应用都要求用户输入，但是大部分都需要。在某些时候，我们需要搜集用户数据。可能是一些状态更新用到的文本，他们的名字和收货地址，搜索条件，他们待办列表的标题等等。
不管这些数据是什么，他都需要用户在模板中输入。为了明确起见，在Ionic 2中我们可以通过以下代码来创建一个表单：

```
<ion-list>

    <ion-item>
        <ion-label>Username</ion-label>
        <ion-input type="text"></ion-input>
    </ion-item>

    <ion-item>
        <ion-label>Password</ion-label>
        <ion-input type="password"></ion-input>
    </ion-item>

</ion-list>
```

以上代码会生成一个简单的登录表单，效果是这样的：



这个登录框允许用户在输入框内输入一些信息。但是，我们需要知道用户在我们的*.html*中输入了什么然后在*.ts*类中使用。本课中我们将讨论一些不同的达成方式。

双向数据绑定

如果你之前用过Ionic 1的话，那么你应该非常熟悉这个概念，如果没有的话，别担心，因为这个改变非常直白。双向数据绑定实际上就是将模板里的一个输入域和类里面的变量值链接起来。以下范例： 模板：

```
<ion-input type="text" [(ngModel)]="myValue"></ion-input>
```

类：

```
myValue: string;

constructor(){

}
```

如果我们改变模板里的，然后类定义里的 *this.myValue* 会被更新到这个值。如果我们改变类里面的 *this.myValue* ，模板里面对于的输入域会自动更新对应的值。通过 *ngModel* 将两个值绑在一起，一个改变的时候，另一个随之改变。
假设我们在模板里有一个提交按钮：

```
<ion-input type="text" [(ngModel)]="myValue"></ion-input>

<button ion-button (click)="logValue()">Log myValue!</button>
```

当用户点击按钮是时候我们将他们输入的值打印到控制台。由于按钮点击的时候会调用*logValue*函数，我们需要在类定义里面添加如下：

```
logValue(){
    console.log(this.myValue);
}
```

不管你在输入框里面输入了什么，这个函数都会拿到然后输出到控制台，当然你也可以用这些数据干更有意义的事情。这是一个非常方便的操作输入的方法，因为我们不需要去担心函数的传入值，我们直接去拿当前的值就可以了。
这在有大量的输入框的时候会显得有点笨拙，所以也不会永远是完美选择。当用于处理复杂表单的时候，我们还有另一个选择，现在就来看另一个选择。

Form Builder

Form Builder是Angular 2提佛那个的一个服务，可以用来更简单的处理表单。Form Builder可以做的事情不止这些，但是他最简单的目的是允许你同时管理大量输入域同时提供了更简单的方法来验证用户输入（例如，是否输入了有效的邮件地址）。
想要使用Form Builder的话，需要先导入（同时导入*FormGroup*）然后注入到你的构造器，如下：

```
import { Component } from '@angular/core';
import { NavController } from 'ionic-angular';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';

@Component({
    templateUrl: 'my-details.html',
})

export class MyDetailsPage {

    constructor(public formBuilder: FormBuilder) {

    }

}
```

请注意，此处也导入了**Validators**，用于与Form Builder验证用户输入。我们来快速了解一下如何使用Form Builder创建一个表单。这个方法最重要的不同是你的输入需要放到定义好了*formGroup*熟悉的

标签里：

```
<form [formGroup]="myForm" (submit)="saveForm($event)">
    <ion-item>
        <ion-label stacked>Field 1</ion-label>
        <ion-input formControlName="field1" type="text"></ion-input>
    </ion-item>

    <ion-item>
        <ion-label stacked>Field 2</ion-label>
        <ion-input formControlName="field2" type="text"></ion-input>
    </ion-item>

    <ion-item>
        <ion-label stacked>Field 3</ion-label>
        <ion-input formControlName="field3" type="text"></ion-input>
    </ion-item>

    <button type="submit">Save Form</button>
</form>
```

也许你注意到了在以上的范例中我们给每个输入域定义了*formControlName*属性，这就是我们现在用作Form Builder区分它们的方式。当然，我们需要一个提交表单的方法，所以我们添加了一个提交按钮同时在表单上添加了一个（submit）监听器用于调用*saveForm*函数。我们稍后定义这个函数，但是为了让所有这些能够正常工作，我们需要在页面的构造器函数里面初始化表单。大概是这样的：

```
this.myForm = formBuilder.group({
    field1: [''],
    field2: [''],
    field3: ['']
});
```

我们只要提交所有表单里面的域（使用我们定义的*formControlName*）然后给所有的域一个初始值（例子中我们使用的是空字符串）。你也可以给每个域提供第二个值来定义Validator，如：

```
this.myForm = formBuilder.group({
    field1: ['', Validators.required],
    field2: ['', Validators.required],
    field3: ['']
});
```

同时注意*this.myForm*变量必须和我们在模板中提供的 *[formGroup]*的值一样。现在来看一下 saveForm 函数：

```
saveForm(event){
    event.preventDefault();
    console.log(this.myForm.value);
}
```

我们传递submit事件然后调用*preventDefault*这样提交表单的默认行为不会发生，我们只想通过这个函数自己处理。我们可以轻松的通过的使用*this.myForm.value*获取用户在表单中的详细数据。

使用Form Builder设置数据稍微复杂些，但是更强大，对于更加复杂的表单值的去尝试。对于更加简单的需求，使用 [(ngModel)] 在大部分案例中更好。

第十课：保存数据

假设你创建了一个Ionic 2应用，在其中用户可以创建购物清单。用户下载了你的应用，花了5分钟添加他们的清单然后关掉应用...然后数据都付诸东流了。
通常制作移动应用的时候你需要去存储数据好让用户稍后获取。大部分时候，我们都是通过将数据存储在远程服务器上（想想Facebook啦，Twitter就是这么干的）并获取实现，当然这个有网络连接需求（下一节课会讲）。在一些案例中，我们想要将数据存储到设备本地。

将数据存放本地是有原因的：

- 应用是完全自容的，与其他用户并无交互，所以所有数据都可以存放本地。
- 我们需要为一些功能存储本地数据，例如记录登录用户
- 我们可以通过本地存储首选设置而无需远程服务器调用
- 我们可以通过本地存储其他数据无需远程服务器调用
- 我们可以同步在线和离线数据因此客户甚至可以在离线的时候持续使用应用（Evernote就是一个典型的例子）

HTML5应用都是运行在浏览器中的，所以我们没有本地应用那种访问本地存储的选择。但是我们还有普通浏览器网页可用的存储选择，**Local Storage**，**Web SQL**（已废弃）和**IndexedDB**。这些选择虽然可能不大理想（理由我会简单说明），但是我们还是可以通过Cordova来弥补浏览器数据存储的不足，通过他访问本地数据存储。
在本地存储数据的选择有很多，但是我们只会讲一些Ionic 2应用的主流做法，也就是**Local Storage**和**SQLite**。

Local Storage

这是谁最基本的存储选择，允许你在用户浏览器中存储**5MB**的数据。记住，Ionic 2应用技术上是运行在内置的浏览器中。
Local Storage封装得有点不好，通常可以认为是不可靠的。我认为浏览器本地存储是一个可行选项，合理的稳定和可靠，但是，数据却可以被擦除，这代表着很多应用不会使用他。即使他在99%的时候好用，但是在存储大部分类型的数据的时候还是不够完美。
总的来讲，你只有在数据丢失无关紧要的时候可以考虑使用它，在存储敏感数据的时候不要用他，因为它很容易访问到。一个很合适的本地存储应用场景是存储一些类型临时会话token。他让你知道是否登录，但是如果数据丢失的话也不是什么大事情，因为用户之需要再次输入用户名和密码再次认证即可。

如果你只是使用本地存储来缓存服务器数据的话，数据也不是什么大问题，因为你会重新从服务器获取。
Local Storage是一个简单的键-值系统，可以通过全局的*localStorage*对象来访问：

```
localStorage.setItem('someSetting', 'off');

let someSetting = localStorage.getItem('someSetting');
```

这是一个设置和获取本地存储数据的本地方法（网页浏览器的本地，不是iOS或者Android本地）。

SQLite

SQLite基本上就是一个可以运行在一个移动设备上的内置SQL数据库。和普通的SQL数据库不一样，他不需要运行到服务器上，不需要任何配置。iOS和Android应用（其他的也可以）都可以用SQLite，但是SQLite数据库只能本地（译者：本地应用的本地，非local，是native）访问，所以他默认是不能被HTML5移动应用访问到的。
我们可以通过Cordova轻松获取这个功能的访问。在项目中运行一下命令就可以安装：

```
ionic plugin add https://github.com/litehelpers/Cordova-sqlite-storage
```

使用SQLite的好处有以下几点：

- 提供了持久的数据存储
- 存储数据无尺寸限制
- 提供了SQL语法，所以是一个强大的数据管理工具

虽然，SQL和SQLite支持的命令有一些区别，但是基本都是一样的。一下是一些SQLite数据查询的范例：

```
db.transaction(function(tx) {
    tx.executeSql('DROP TABLE IF EXISTS test_table');
    tx.executeSql('CREATE TABLE IF NOT EXISTS test_table (id integer primary key, data
 text, data_num integer)');

    tx.executeSql("INSERT INTO test_table (data, data_num) VALUES (?,?)",   ["test", 1
00], function(tx, res) {
        console.log("insertId: " + res.insertId + " -- probably 1");
        console.log("rowsAffected: " + res.rowsAffected + " -- should be 1");
    }, function(e) {
        console.log("ERROR: " + e.message);
    });
});
```

上面的例子看起来相当怪异，但是如果你熟悉SQL的话，至少其中一些你看着眼熟。这是标准的通过Cordova使用SQLite的方法，但是Ionic提供了他自己的使用SQLite的服务。

Ionic Storage

幸运的是，对于大部分存储需要我们不用担心实现细节。Ionic提供了他自己的存储服务，可以自动使用最佳的存储机制（不管它是老的本地存储，Web SQL， IndexedDB或者SQLite）。无论背后使用的是什么存储机制，他始终给我们提供统一的API。
想要使用的话，先知**app.module.ts**文件中导入和添加：

```
import { Storage } from '@ionic/storage';
```

然后添加到提供者数组：

```
providers: [Storage]
```

然后你可以注入到任何你要用到的类中。以下面的提供者为例：

```
import { Storage } from '@ionic/storage';
import { Injectable } from '@angular/core';

@Injectable()
export class Data {

    constructor(public storage: Storage){

    }

    getData(): any {
        return this.storage.get('checklists');
    }

    save(data): void {
        let newData = JSON.stringify(data);
        this.storage.set('checklists', newData);
    }
}
```

我们只要简单的调用*this.storage.set*就可以将数据存储到指定的键（这里是*checklists*）上去，后面我们利用*this.storage.get*方法传入这个键来获取这些数据。
如果你的数据存储要求更复杂，这可能不是你的完美之选，但是对于很多场景来讲，这个提供了一个很简单的存储数据的方法。

第十一课：获取数据，Observable和Promise

虽然很多移动应用是完全自容的（计算器，音板，待办列表，照片应用，手电筒应用），很多应用靠着从外部源拉取数据。Facebook需要为新闻种子拉取数据，Instagram的最新图片，天气原因的最新天气预报等等。
这个部分我们将涵盖如何拉取外部数据到你的Ionic应用。在进入具体学习从外部服务获取数据之前，我得先涵盖一点我们将要做什么的知识。

映射与过滤数据

map映射和**filter**过滤功能非常强大，允许你对数组做很多事情。这些不是神奇的ES6或者Angular 2功能，他们是JavaScript的一部分。
简单起见，映射针对数组的每个数据，对他运行一些函数，这些函数可能会改变数据的值，然后将他放到一个新的数组中。他将一个数组的每个值映射到一个新的数组。为了给你一个直观的理解，以下范例展示他为何有用，假定你有一个文件名的数字，这样：

```
['file1.jpg', 'file2.png', 'file3.png']
```

然后你就可以将这些值的完整路径映射到一个新的数组：

```
['http://www.example.com/file1.jpg', 'http://www.example.com/file2.png','http://www.example.com/file2.png']
```

代码是这样的：

```
let oldArray = ['file1.jpg', 'file2.png', 'file3.png'];

let newArray = oldArray.map((entry) => {
    return 'http://www.example.com/' + entry;
});
```

我们给**map**提供了一个函数用于返回修改的值。我们提供的函数会接受每一个值作为传入参数。

过滤和映射很像，但是和映射到一个新的数组不同，他只会添加匹配指定标准的值到新的数组。我们就用之前的例如，但是这次我们想要返回所有包含*.png*文件的数组。代码如下：

```
let oldArray = ['file1.jpg', 'file2.png', 'file3.png'];

let newArray = oldArray.filter((entry) => {
    return entry.indexOf('.png') > -1;
});
```

如果我们同时还要完整路径的话，我们可以将**filter**和**map**链接起来用：

```
let oldArray = ['file1.jpg', 'file2.png', 'file3.png'];

let newArray = oldArray.filter((entry) => {
    return entry.indexOf('.png') > -1;
}).map((entry) => {
    return 'http://www.example.com/' + entry;
});
```

这样，我们就先过滤出我们不需要的结果，然后给他们映射完整路径到一个新的数组。结果将是一个包含了完整路径的两个*.png*文件的数组。这个就先停止这里，当我们学习获取数据的范例的时候，为什么值得花费这么多精力去解释他将会很明了。

Observables 和 Promises

如果你用过Ionic 1或者有很强劲的JavaScript背景的话，那么你可能熟悉**Promises**，但是熟悉**Observables**的人非常少。Observable核心新功能之一，包括Angular2（RxJS提供），所以理解他是什么他们和Promise的区别非常重要（他们看起来和表现起来都很像）。
在进入Observable之前，我们先具体了解一下Promise是什么。当我们编写异步（意思是代码不是顺序执行的）代码的时候，Promise就介入了。在使用HTTP请求数据的时候，我们需要等待数据返回，由于这个返回可能需要等待1-10秒钟，我们不可能在这个等待阶段同步停止整个应用。我们需要保持应用中等待的同时一直运行接受新的用户输入，当最终得到请求返回的时候，我们再对他进行处理。
Promise就是用来对付这种情况的，如果你熟悉回调函数，那么它就是相同的理念，只是更好一些。假设我们有一个方法名为*getFromSlowServer()* 返回一个promise，我们这样去使用：

```
getFromSlowServer().then((data) => {
    console.log(data);
});
```

我们调用Promise提供的*then*方法，这个方法是意思就是“已收到服务端返回数据，用他来处理”。在这个例子中，我们将返回数据传入一个将他打印到控制台的方法。这样，我们的应用就可以去做别的事情了，当数据返回时他就会执行以上代码。你可以想想爱你个你在工作并在写报告，你需要一些其他信息，所以你让你的助手去帮你找，但是你不需要坐在那里等你

的助手回来 -- 你继续写你的报告然后在你的助手回来的时候你就用上那些信息。
我们现在知道Promise是什么了，然后Observable是什么他能做什么Promise做不到的？
Obeservable和Promise服务的是同一个目标，但是他做了一些额外的事情。最主要的不同是Promise返回单纯的数据，但是一个**Observable**是一个流（**stream**）可以多次收到数据。将Observable看作流更简单，因为他们本身就是，他们叫做Observable是因为他们是可观测的（因为我们可以检测从流收到的数据）。
Observable看起来和Promise很想，但是他是用*subscrible*而不是*then*。由于Promise只是返回纯粹的值，所以在数据返回的时候可以直接使用。我也说过，Observable是一个流可以收到很多值，所以订阅它（和你喜欢的Youtube频道一样）也很合理，在每次收到数据时运行一些代码。看起来大概是这样的：

```
someObservable.subscribe((result) => {
    console.log(result);
});
```

很明显在进行HTTP请求的时候我们的应用需要等待数据返回，因此Promise和Observable非常有用。这不是唯一需要异步编程的例子。有一些不那么明显的场景例如从获取本地存储数据，甚至从用户相机获取相片，这些场景都需要等待处理结束才能使用具体数据。
如果想深入以上讨论的东西，强烈推荐交互手册。他介绍了RxJS，里面包括了Observable，也建立了如何使用**map**，**filter**和其他功能的坚实基础。如果想更具体的了解Observable与Promise的不同，强烈推荐egghead.io视频。

利用Http从服务端拉取数据

好了，你可能已经有了理论只是的武装 -- 我们来学习一个例子。这里我要用Reddit API来举例，因为他是一个公开访问且非常同意使用的接口。如果你订购了本书的包包括Giflists应用，那么我们后面会更详细的了解这个。
你可以简单的通过访问以下格式的URL来从subreddits创建一个帖子的JSON反馈（feed）：
https://www.reddit.com/r/gifs/top/.json?limit=10&sort=hot

如果你点击以上链接，你会看到一个从**gifs**subreddit返回的返回的JSON反馈，包括10个子任务，以热度排序。如果你不熟悉JSON的话，我建议你阅读以下这个 -- 但是实际上它叫做*JavaScript Object Notation*是一个很好的传输数据的方式因为他可读性很好，电脑解析也很容易。如果你之前这样创建过JavaScript对象：

```
var myObject = {
    name: 'bob',
    age: '43',
    hair: 'purple'
};
```

那么你稍微整理一下就可以很轻松的阅读JSON反馈。但是我们要怎样将他带入到我们的Ionic 2应用中呢？
答案是使用Angular 2提供的**Http**服务，他允许你发起HTTP请求。如果你不知道HTTP请求是什么，基本上浏览器每次加载任何东西（文档，图片，文件等等）的时候，他就会发起HTTP请求。所以我们可以发起一个页面的HTTP请求，然后可以返回一些JSON数据给我们的应用使用。
首先我们需要设置**Http**服务，我们来看一个测试页面导入并注入了这个服务：

```
import { Component } from '@angular/core';
import { Http } from '@angular/http';
import 'rxjs/add/operator/map';

@Component({
    selector: 'page-one'
    template: 'page-one.html'
})

export class Page1 {
    constructor(public http: Http) {

    }
}
```

由于咱们注入了Http服务，通过*public*修饰符我们可以使用*this.http*来自类里面使用。同时注意，我们从RxJS库里面导入了map操作符。之前讲过map是数组默认提供的函数 -- 所以为什么我们需要从一个奇怪的库里面导入呢？因为Http服务并不会返回一个数组，他返回的是**Observable**。RxJS库为Observable提供了map函数，所以我们的先进行导入。
现在我们看一下想reddit URL发起请求的代码：

```
import { Component } from '@angular/core';
import { Http } from '@angular/http';
import 'rxjs/add/operator/map';

@Component({
selector: 'page-one'
template: 'page-one.html'
})

export class Page1 {
    constructor(public http: Http) {
        this.http.get('https://www.reddit.com/r/gifs/new/.json?limit=10').map(res => r
es.json()).subscribe(data => {
            console.log(data);
        });
    }
}
```

调用的第一部分给咱们返回一个Observable。然后我们使用**map**函数，在其总将JSON文本通过*json()*函数将他对象化。这样我们就能更好的去用了。
重要：记住，Http请求是异步的。意思是你的代码只会在拉取到数据之后才会向前执行，这里过程可能是几毫秒到10秒，甚至不会返回。所以应用得预先做好应对。如果你运行如下代码：

```
this.posts = null;
this.http.get('https://www.reddit.com/r/gifs/top/.json?limit=2&sort=hot').map(res => r
es.json()).subscribe(data => {
    this.posts = data.data.children;
});

console.log(this.posts);
You would see null output to the console. But if you were to run:
this.posts = null;
this.http.get('https://www.reddit.com/r/gifs/top/.json?limit=2&sort=hot').map(res => r
es.json()).subscribe(data => {
    this.posts = data.data.children;
    console.log(this.posts);
});
```

你可以看到帖子输出到控制台，因为**subscribe**函数里面的所有事物都会在数据返回的时候运行。
返回我们的例子，在我们映射完响应之后我们链了一个subscribe调用在其中我们可以使用通过流（Observable）提交来的数据。之前说过Observable有用因为我们可以一直监听不同的值...但是为什么在这里用他？Http调用只会返回一次结果，为什么不用Promise替代Observable来避免混淆大家视听？
此处偏爱Observable而不是Promise的原因是Observable可以做Promise做的所有，在场景外技术上更好，可以做Promise做不到的其他事情。例如我们可以设置一个定时器**interval**来每5秒或者10秒发起Http请求，我们可以简单的设置**debouncing**保证请求不会太频繁的发起（预防向服务器大量发起请求），如果在旧请求结果返回之前Observable可以取消'在发'请求等等。
以下是Giflists应用中的代码：

```
this.subredditControl.valueChanges.debounceTime(1000)
  .distinctUntilChanged().subscribe(subreddit => {
    this.subreddit = subreddit;

    if(this.subreddit != ''){
        this.changeSubreddit();
    }
});
```

在这里*subredditControl*是一个**Observable**。用户使用应用里输入就可以控制他的值。**subscribe**里面的代码只会在1秒（debounceTime）内不在变动的情况和新值和之前的值不一样的情况下执行。这个例子可以很清晰的表达**subscribe**帮你处理了多少奇怪的东西。
如果看到上面的例子你想'哇... Ionic 2太难了，我要回去找Ionic 1！'，最好不要！这是举证Observable的一个高级范例，Ionic 2输入处理和双向数据绑定和Ionic 1一样简单。
Observable是一个大的主题，还有成吨的东西需要去学习。如果你不知道啥情况的话也不要被吓到。在制作Ionic应用的时候最好对Obserable有一个比较好的认识，但是对于Ionic来讲，他只是理解就好（虽然他比较重要但是你不是经常用到）你同样可以过的很好。

从自己的服务器拉取数据

我们知道了如何使用JSON反馈从类似reddit提供的服务器拉取数据，但是如果你想拉取自己的数据呢？你要怎样才能设置你自己的JSON反馈呢？
详细深入如何设置你自己的API超出了本课程的范围，但是我会给你指出大概。基本上：

- 1.在Ionic应用里面发起请求到你的服务器URL
- 2.从你的服务器拉取数据，服务器语言随你喜欢
- 3.以JSON格式将需求的数据打印到页面

我会带你通过PHP实现一个简单的API，-- 你可以用你自己喜欢的语言 -- 这样你就可以输出JSON到浏览器：

- 1.创建一个文件名为*feed.php*，文件可以通过 *http://www.mywebsite.com/api/feed.php* 访问到
- 2.获取数据。本案例中我通过查询MySQL，当然数据来源可以是不同的方式：```php $mysqli = new mysqli("localhost", "username", "password", "database"); $query = "SELECT * FROM table"; $dbresult = $mysqli->query($query);

while($row = $dbresult->fetch_array(MYSQLI_ASSOC)){ $data[] = array('id' => $row['id'], 'name' => $row['name']); }

if($dbresult){ $result = "{'success':true, 'data':" . json_encode($data) . "}"; } else { $result = "{'success':false}"; }

```
- 3.将数据JSON化输出到浏览器：```echo($result);
```

- 4.在应用中通过*http.get()*请求 *http://www.mywebsite.com/api/feed.php*。

之前讲过，可以使用任何服务端语言，可以使用任何存储机制。只要能够以JSON格式获得你要的数据，然后输出到浏览器就可以了。

这个给了你一个很合理的概观：如何使用Ionic 2 和Http服务来获取远程数据。这里的代码第一眼看来语法和概念可能有些花俏，但是当你进入基础的时候会发现你需要知道的不是太多。你的数据可能更复杂，你也许想要对他进行更神奇的操作，或者以不同的方式在显示，但是基本原理还是一致的。

> 译者：这两章翻译得哟点烂，因为心情焦躁

第十二课：本地功能

基于网页的移动应用的问题在于我们可以通过浏览器在iOS和Android上运行，用户不能将它们安装在设备本地，应用不能访问本地API，如：联系人，蓝牙等等。这就是为什么我们使用**Cordova**结合Ionic，Cordova允许我们将应用打包为一个本地封装，这样允许我你们提交到应用商店，同时也可以通过插件来使用本地API。用来Cordova之后，HTML5移动应用可以做到本地应用能做的一切。
刚才也说到了，想要使用本地功能的话我们就要用到插件。Cordova提供了大量的默认插件包括：

- 设备 Device
- 网络信息 Network Information
- 相机 Camera
- 地理信息 Geolocation
- 文件 File
- 应用内浏览器 In App Browser
- 多媒体 Media
- 启动画面 Splash Screen

同时，基本上所有的功能都有社区开发的开源插件，以下是一些比较流行的：

- 本地通知
- Facebook连接
- SQLite
- 社交分享

插件所做的基本上就是创建一个接口给JavaScript触发本地功能的调用。所以，如果你需要用到一个不存在的Cordova插件的时候（基本上不会遇到这样的情况），你可以自己去写（当然也需要写本地代码）。
重要：大部分插件只在真实设备上才可以使用，如果你想通过ionice serve去测试Cordova插件的时候可以能会出现错误。

在Ionic 2中使用Cordova插件

Ionic 2中实现本地功能有两种方法。可以通过安装Cordova插件直接使用：

```
ionic plugin add plugin-name
```

然后访问插件提供的本地功能，通常可以通过全局对象去访问：

```
window.plugins.somePlugin.someMethod();
```

不需要导入，不需要需求（required），不需要在特定地方调用或者不需要其他任何东西 -- 一旦你通过命令行安装好了插件之后你可以在任何地方访问。并不是所有的插件都可以这样去访问，但是大部分都是这么用的。这不是Ionic 2特有的，任何Cordova项目（唯一的不同是使用*cordova plugin add*，而不是*ionic plugin add*）都可以这样去使用。当使用正常Cordova语法的时候，在Ionic 1，Ionic 2，Sencha Touch， jQuery Mobile或者其他使用Cordova的网页里面使用插件并没有不同。
记住如果你这样去使用Cordova插件的话，你的应用可能由于TypeScript警告而编译不了。这是因为TypeScript不知道他是什么，可能你需要去安装他对应的*typings*。如果想强行通过编译的话，你可以简单的加上：

```
declare var variableCausingProblems;
```

在类定义的装饰器上面添加你用到的插件。
另一中方法，你可以用**Ionic Native**来使用Cordova插件，这是Ionic 2特有的。如果你对Ionic 1的ngCordova很熟悉，这基本就是Ionic 2里面的同一个东西。如果你不熟悉ngCordova的话，Ionic Native基本上就是通过Promise和Observables让Cordova插件更好的与Angular 2工作。
Ionic Native在Ionic 2应用中是默认安装的。所以你需要做的就是像平常那样安装你需要用到的插件，像这样：

```
ionic plugin add cordova-plugin-geolocation
```

接下来，你需要在要用到这个插件的类里面通过Ionic Native导入此插件：

```
import { Geolocation } from 'ionic-native';
```

然后你就可以在代码里面使用了：

```
Geolocation.getCurrentPosition().then((resp) => {
    console.log("Latitude: ", resp.coords.latitude);
    console.log("Longitude: ", resp.coords.longitude);
});
```

注意，上面代码返回了一个promise，我们通过.*then()*设置了一个操作器（handler），如果我们使用标准的Cordova语法的话，那么是用不了这个的 -- Cordova的标准语法是使用回调函数，会比较乱。
同时需要中哦你大概ianjizhud是不是所有的Cordova插件都能通过Ionic Native使用。所有可用的组件以及如何使用，请查看Ionic Native 文档。如果你想用的组件中Ionic Native里面没有

的话，那么你只能回头使用标准Cordova语法了（或者你可以自己动手添加到Ionic Native）。
虽然不是强制要求，你可以在任何地方使用Ionic Native。他可能使你的代码更整洁，更是个Angular 2生态系统（typings自动处理这些，所以TypeScript不会抱怨）。用老式的Cordova不会死人，所以也不要觉得他太糟糕。

第三章：快捷列表 Quick Lists

第一课：介绍

快捷列表是实际的本课程的手把手操作应用 -- 不论你购买的是哪个版本，你都会的有这个课程。我选择快捷列表来填充整个角色是因为他涉及到了Ionic 2的大部分的核心概念，创建此应用学习的技能将会在后续课程的应用中频繁用到。
大部分人（包括我自己）在解释一些新技术或者框架的时候都会选择制作一个todo应用制作教程，原因是todo应用涵盖了大部分基本知识，例如：

- 总体架构 & 设置
- 用户界面
- 数据的增删改查
- 接受用户输入

很明显，这些都是需要去重点学习的概念，但是本书中我真的不想去创建一个todo应用了 -- 我想做一些稍微复杂和有趣一点的东西。结果基本跟todo相同，也涵盖了todo应用差不多的基础内容，但是我觉得制作这个会更有趣一些。

关于快捷列表

快捷列表的想法是来自我自己的个人需求。写这段文字的时候我正在远程工作同时坐着我的大篷车环游澳大利亚（译者：羡慕啊!_!）。这当然是一个非常好的体验，实际上类似每周拖着你的房子游历全国（好大的一个国家）也会有些不大好的事情。
一个特别复杂的事情是打包蓬子然后挂到车子上去，卸包和安装蓬子也一样。我不会跟你讲那些枯燥的细节来烦你，但是每次会有最少20种待办和待查事宜。有些是无关紧要的，但是有些则非常重要，例如确定链子是否栓好在车子上，天然气存量是否足够，刹车和指示盘是否正常。
所以我觉得制作一个原因用来创建“起航前”检查列表。检查列表包含了大量的需要检查是否完成的事情，当你下次需要重新开始这个检查列表的时候，点击一个刷新按钮就可以重置所有事情。也就是一个可重复的待办列表应用。
如我所言，这个应用覆盖了传统待办列表应用相同的概念。这些改变包括：

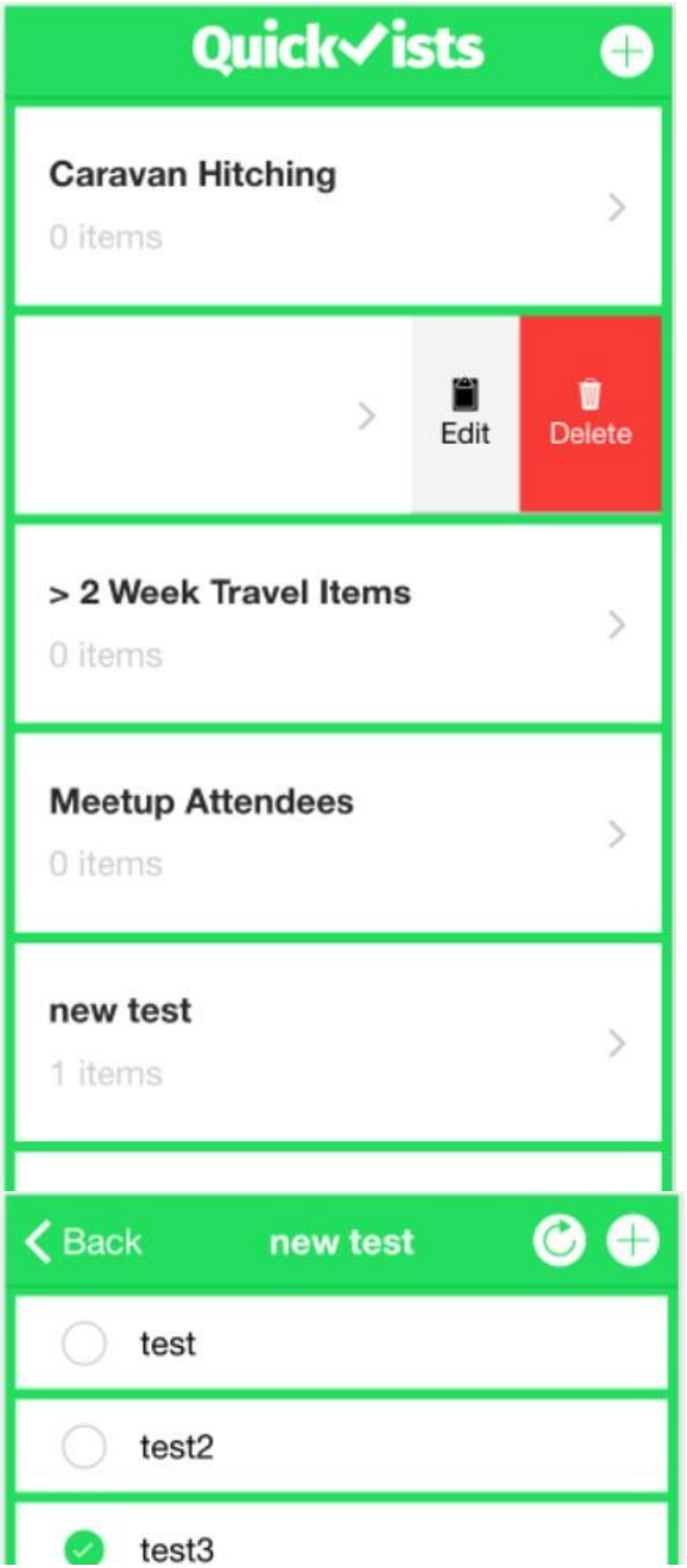
- 复杂列表
- 数据模型
- Observable
- 表单和用户输入
- 简单导航
- 页面之间传递数据
- 数据的增，删，改，查
- 数据存储和获取

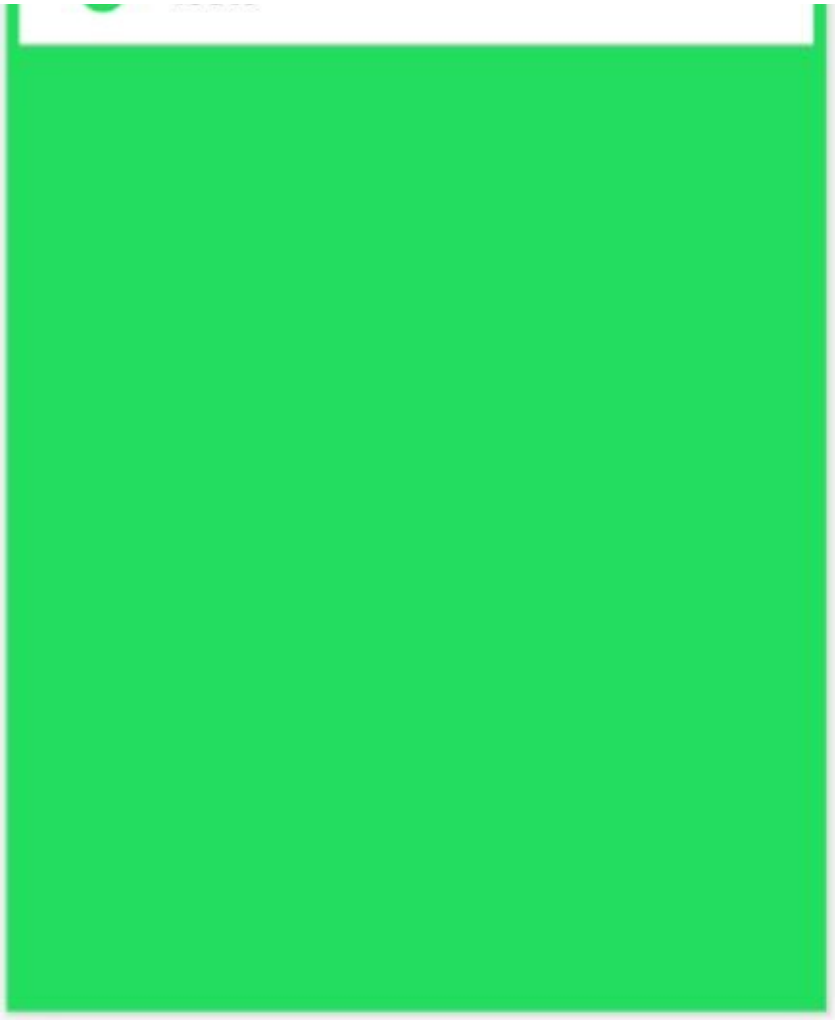
- 定制主题

以下是由于的具体功能：

- 第一次打开应用会展示一个介绍手册
- 用户可以创建任意数量的检查列表
- 用户可以给任何检查列表添加任意数量的条目
- 检查列表里面的条目可以被设置为完成和未完成
- 用户可以在任何时候“重置”一个检查列表
- 用户可以编辑或者删除任何的检查列表或者检查列表里面的条目
- 在返回应用的时候所有数据可以被回忆起来（包括所有条目的完整状态）

以下是内容的截屏：


Quick✓ists
Caravan Hitching
0 items
Edit
Delete
> 2 Week Travel Items
0 items
Meetup Attendees
0 items
new test
1 items
Back
new test
test
test2
test3



课程结构

1. 准备工作
2. 基本布局
3. 数据模型和Observables
4. 创建检查列表和列表项
5. 保存和加载数据
6. 制作一个介绍滑页 & 自定义主题

准备好了吗？
现在你知道你要干什么了，那么我们开始吧！

第二课：准备工作

本课中山我们在旅程继续之前需要的一些准备工作。我们当然是要生成应用，我们当然也要设置好所有组件和需要用到的Cordova插件。
在开始创建新应用之前的首要工作是确定你是否是最新的Ionic和Cordova版本，如果你最近没有做过的话，那么在继续之前运行一下以下命令：

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

或者

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

如果你在安装Ionic或者生成新项目的时候遇到任何问题，请先确定一下你安装了最新版本的Node。安装完成之后，如果你要重新安装一遍的话，那么请先运行以下命令：

```
npm uninstall -g ionic npm cache clean
```

生成一个新应用

我们将使用空白开始模板（blank starter template）也就是生成一个空的Ionic项目。它会生成一个内置的页面叫做**home**，也就是我们的主页，下一课里面用于持有检查列表。
> 运行如下命令生成一个新项目

```
ionic start quicklists blank --v2
```

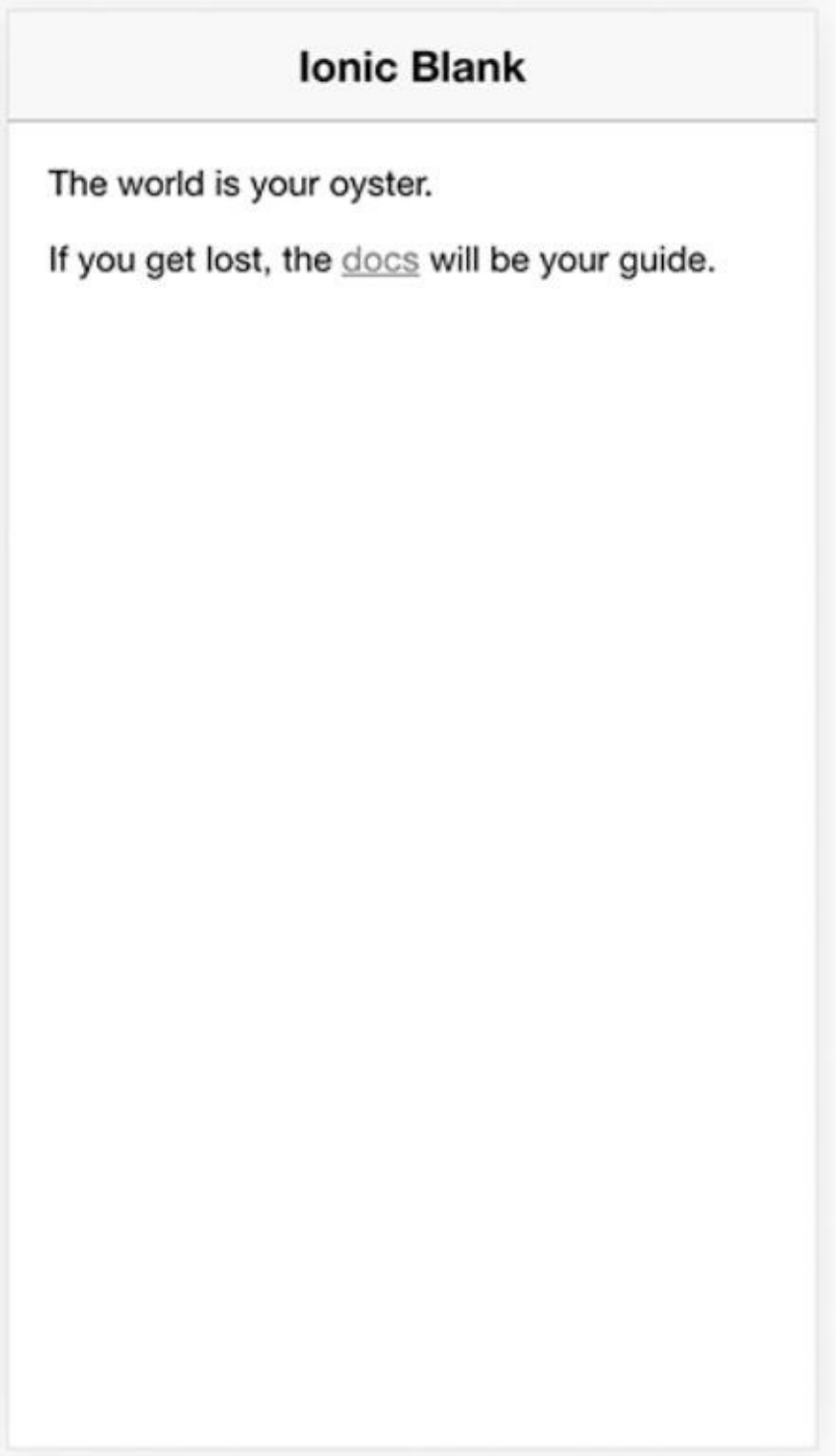
> 通过如下命令将项目目录作为当前目录

```
cd quicklists
```

这样，你的新项目就生成了 -- 你就可以在你喜欢的编辑器里面打开这个项目了。运行如下命令可以预览你生成的应用：

```
ionic serve
```

现在你应该可以看到以下画面：



创建需要的组件

这个应该需要用到总共3个页面组件。我们有**HomePage**用来展示所有检查列表的列表，一个**IntroPage**用来展示介绍手册，一个**ChecklistPage**用来展示指定检查列表的列表项。我们已经有了Home页面，所以我们只需要创建其他的两个就可以了。
> 运行如下命令生成介绍页面：

```
ionic g page Intro
```

> 运行如下命令生成列表详情页面：

```
ionic g page Checklist
```

创建需要的服务

我们创建一个数据服务。他将用于操作保存检查列表数据到存储和从存储中获取数据。
> 运行如下命令生成数据提供者：

```
ionic g provider Data
```

创建数据模型

我们将要给我们的检查列表创建数据模型，以允许我们更简单的创建和更新数据。不幸的是，没有可以用来生成数据模型的命令，我们得手动创建。
> 在**src**文件夹内创建一个的文件夹名为**models**
> 在**models**文件夹内新建一个文件名为**checklist-model.ts**

在App Module里面添加 页面 & 服务

为了能够在项目里面可以使用这些页面和服务，我们需要将它们添加到**app.module.ts**文件里。所有我们自己创建的页面都需要添加到*declarations*数组和*entryComponents*数组里，所有我们创建的数据提供者都需要添加到*providers*数组，其他自定义组件或者管道（pipe）只需要添加到*declarations*数组即可。我们的数据模型只是一个简单的类，我们需要在任何地方使用，所以不用在模组里面设置。

> 修改**src/app/app.module.ts**到以下：

```
import { NgModule } from '@angular/core';
import { IonicApp, IonicModule } from 'ionic-angular';
import { MyApp } from './app.component';
import { Storage } from '@ionic/storage';
import { HomePage } from '../pages/home/home';
import { IntroPage } from '../pages/intro/intro';
import { ChecklistPage } from '../pages/checklist/checklist';
import { Data } from '../providers/data';

@NgModule({
    declarations: [
        MyApp,
        HomePage,
        IntroPage,
        ChecklistPage
    ],
    imports: [
        IonicModule.forRoot(MyApp)
    ],
    bootstrap: [IonicApp],
    entryComponents: [
        MyApp,
        HomePage,
        IntroPage,
        ChecklistPage
    ],
    providers: [Storage, Data]
})
export class AppModule {}
```

注意，我们除了自己创建的*Data*提供者之外，我们还加入了一个*Storage*。Storage是Ionic提供的，可以通过它保存和获取数据 -- 我们后续会用到。

添加需要的平台

在给指定平台制作应用之前，你需要将它们添加到你的项目。
> 运行以下命令添加**iOS**平台：

```
ionic platform add ios
```

> 运行以下命令添加**Android**平台：

```
ionic platform add android
```

添加需要的Cordova插件

这个应用将会用到不同的Cordova插件。记住，Cordova插件只能在真实设备上运行。我将在添加他们的时候解释。
> 运行以下命令添加**SQLite**插件：

```
ionic plugin add cordova-sqlite-storage
```

这个插件让你可以访问本地存储SQLite数据库。我们在此应用中添加他的原因是Ionic本地存储服务可以使用插件提供的稳定输出存储。
> 运行以下命令添加**Status Bar**插件：

```
ionic plugin add cordova-plugin-statusbar
```

我们给所有项目添加此插件用来在应用中控制状态栏（设备屏幕顶部的状态条，包括时间，电池信息等等）。
> 运行以下命令添加**Splash Screen**插件：

```
ionic plugin add cordova-plugin-splashscreen
```

此插件允许我们控制闪屏（打开应用的时候的全屏画面）。
> 运行以下命令添加**Keyboard**插件：

```
ionic plugin add ionic-plugin-keyboard
```

这个插件允许我们控制软键盘。
> 运行以下命令添加**Whitelist**插件：

```
ionic plugin add cordova-plugin-whitelist
```

所有应用会用到这个插件，他定义了应用里可以加载什么样的资源。没有他的话，你尝试加载的资源都会不成功。
添加了这个插件后，你也需要到**index.html**中定一个一个“Content Security Policy”。我们将添加一个非常宽松的策略实际上允许我们加载任何资源。基于你的应用，你可以提供一个更严格的策略，但是对于开发而言开放性策略就可以了。
> 修改 **src/index.html**文件，添加一下**meta**标签：

```
<meta http-equiv="Content-Security-Policy" content="font-src 'self' data:;
img-src * data:; default-src gap://ready file://* *; script-src 'self'
'unsafe-inline' 'unsafe-eval' * ; style-src 'self' 'unsafe-inline' *">
```

> 运行以下命令添加**Crosswalk**插件：

```
ionic plugin add cordova-plugin-crosswalk-webview
```

这个另一个每个应用都要添加的插件，但是你也可以先不添加。添加了这个插件后，在你编译Android的时候将会使用“Crosswalk”。Android有很多问题，特别是老设备，因为有太多不同的团建版本，不同的版本有不同的浏览器（记住，鉴于我们是制作HTML5应用，他实际上就是一个搭载的浏览器用来运行我们的应用）。Crosswalk做的是将一个现代的浏览器打包到应用中，这样一来应用无论是运行在什么设备上，都会使用相同的浏览器来运行，并且Crosswalk浏览器可以很好改善执行效率。
唯一的不足之处就是你的应用尺寸明显的变大了很多。总体上，我觉得这很值得，我也建议你使用他，如果你接受不了的话，也可以不用。更多关于Crosswalk Project的信息，请参考网站：https://crosswalk-project.org/

设置图片

制作此应用的时候，会用到一些图片。你下载的包里面已经包含了这些图片，但是你需要去生成的项目里面设置好他们。
> 将下载包 **src/assets**文件夹下面的**images**文件夹复制到应用里的 **src/assets**下面

总结

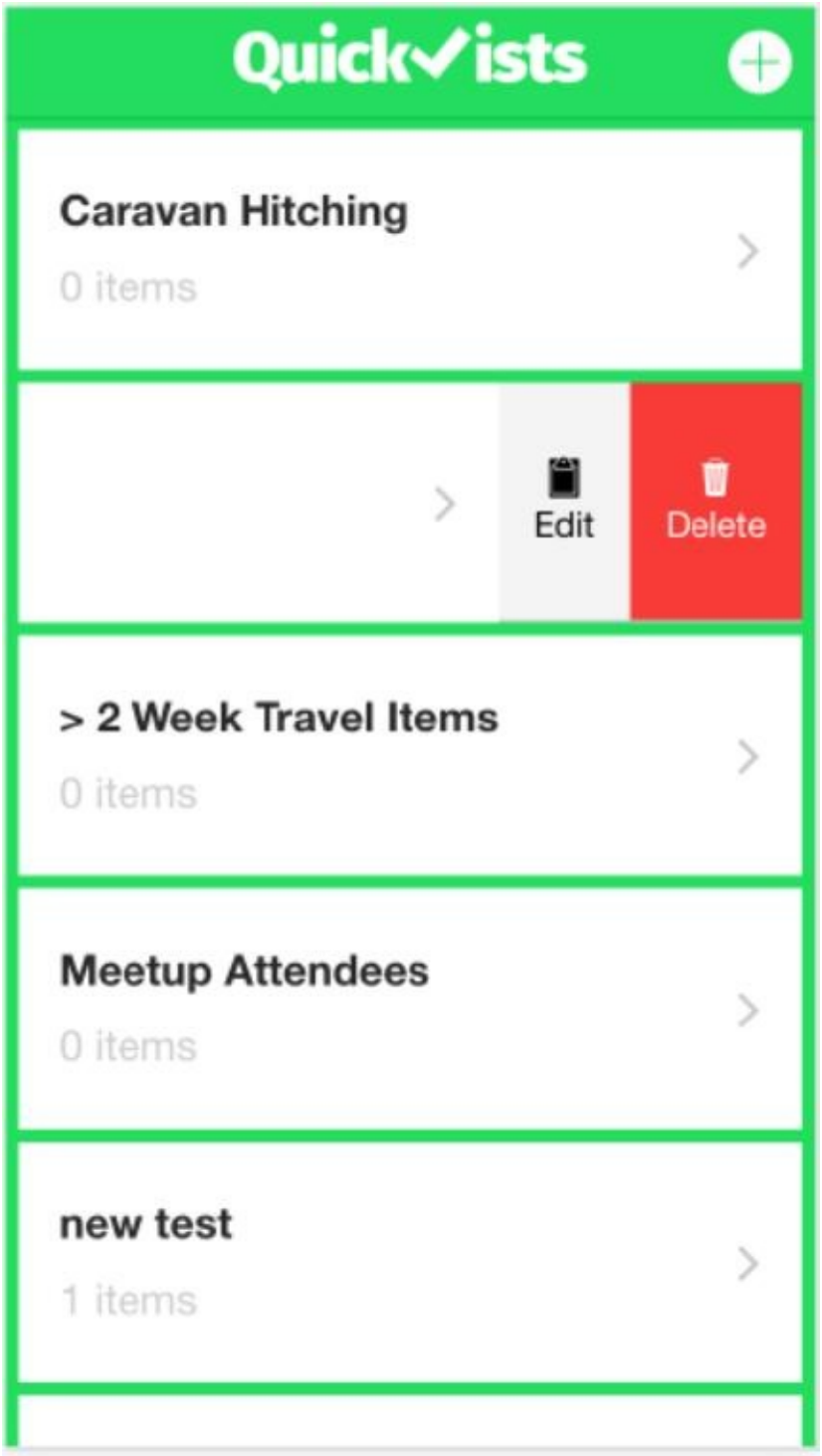
就这样！我们设置好了，准备好继续前几，现在我们开始进入到有趣的部分了。

第三课：基本布局

课程开始我们学习会很慢且很简单，只是专注创建应用的基本布局。我们需要给**Home**页面创建模板，用来显示所有创建的检查列表，对于**Checklist**页面，用于展示一个指定检查列表的所有条目。如果你稍加留意的话，你就知道还有另一个页面，这个的制作就是比较靠后的事情了。
如之前所说，我会尽量使课程模组化，这样制作应用可以引起你的兴趣，而不用强制遵循一个固定的顺序。由于这是第一个应用，也因为这是基本包里面唯一包含的原因，我会注意确保所有细节解释清楚。

Home页面

在进入代码编写之前，我们需要在脑中有一个清晰的视图我们要制作什么样子的一个东西（译者：胸有成竹）。一下是完成的home页面的截图：



可以看到它有一个很漂亮的样式，这个我们后续会涉及，但是实际上他就是个很简单的列表，右上角有一个按钮用来添加一个新的检查列表。虽然不是全部都简单，你会发现其中一个列表项有点不一样，他有一个‘Edit’和‘Delete’按钮。这是因为我们将使用Ionic提供的滑动列表组件，他允许我们指定在用户滑动列表项的时候展示一些内容。

那么，我们开始制作吧。首先，我们看一下整个模板里面的内容，然后我们将他分成小块来详细讨论： **>** 修改**src/pages/home/home.html**为如下内容

```
<ion-header>
    <ion-navbar color="secondary">
        <ion-title>
            <img src = "assets/images/logo.png" />
        </ion-title>
        <ion-buttons end>
            <button ion-button icon-only (click)="addChecklist()"><ion-icon name="add-
circle"></ion-icon></button>
        </ion-buttons>
    </ion-navbar>
</ion-header>

<ion-content>
    <ion-list no-lines>
        <ion-item-sliding>
        <button ion-item (click)="viewChecklist(checklist)">
        TITLE GOES HERE
        <span>0 items</span>
        </button>
            <ion-item-options>
            <button ion-button icon-only color="light" (click)="renameChecklist(checkl
ist)"><ion-icon        name="clipboard"></ion-icon> Edit</button>
            <button ion-button icon-only color="danger"  (click)="removeChecklist(chec
klist)"><ion-icon
            name="trash"></ion-icon> Delete</button>
            </ion-item-options>
        </ion-item-sliding>
    </ion-list>
</ion-content>
```

我们先开始讨论部分：

```
<ion-header>
    <ion-navbar color="secondary">
        <ion-title>
        <img src = "assets/images/logo.png" />
            </ion-title>
        <ion-buttons end>
            <button ion-button icon-only (click)="addChecklist()"><ion-icon   name="ad
d-circle"></ion-icon></button>
        </ion-buttons>
    </ion-navbar>
</ion-header>
```

允许我们添加一个页首条到我们的应用，可以持有一些按钮，标题甚至在需要的时候直接与lonic的导航系统整合来显示后退按钮。
我们给navbar添加了color属性并制定为secondary颜色，此颜色定义在**theme/variable.scss**里面。navbar里面我们用到一个，他基本上就是用来展示一个当前页面的文本标题，展示

logo。我们也用到了在navbar里面创建了一个按钮。通过指定end属性，所有按钮在iOS上都会排列在右边，但是如果我们指定的是'start'属性，按钮将会排列到左边。记住，Ionic 2有平台一致性编译，所以在不同的运行平台上，他默认会展示到合适的位置上。
最后，我们将按钮放到里面。这个按钮用到了一个圆形图标并添加了一个点击处理器在点击的时候调用**home.ts**里面的*addChecklist()*方法（现在还没创建这个方法）。同时注意，我们给按钮用了*ion-button*和*icon-only*属性，这可以是Ionic知道我们将对按钮使用Ionic样式，这个样式将使按钮只有图标没有文本。
我们现在来看列表部分：

```
<ion-content>
    <ion-list no-lines>
        <ion-item-sliding>
            <button ion-item (click)="viewChecklist(checklist)">
            TITLE GOES HERE
            <span>0 items</span>
            </button>
            <ion-item-options>
            <button ion-button icon-only color="light"   (click)="renameChecklist(chec
klist)"><ion-icon
            name="clipboard"></ion-icon> Edit</button>
            <button ion-button icon-only color="danger"  (click)="removeChecklist(chec
klist)"><ion-icon   name="trash"></ion-icon> Delete</button>
            </ion-item-options>
        </ion-item-sliding>
    </ion-list>
</ion-content>
```

在讲解列表之前，我们先看到所有的东西都被包围在里面 -- 这个标签用来持有页面的主体内容，在大部分的案例中，所有navbar之外的内容都将放在这里。
和纯HTML创建的列表相比：

```
<ul>
    <li></li>
    <li></li>
    <li></li>
</ul>
```

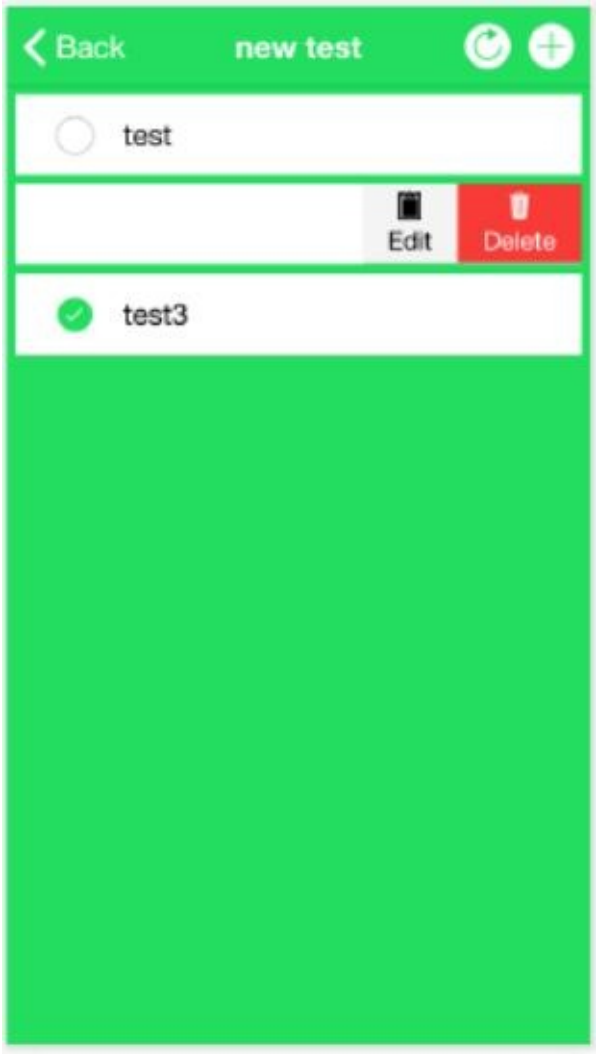
Ionic创建列表的方式基本上是一样的：

```
<ion-list>
    <ion-item></ion-item>
    <ion-item></ion-item>
    <ion-item></ion-item>
</ion-list>
```

当然，我们的看起来稍微复杂些，所以我们来了解一下。第一个超出寻常的事物是我们家了一个 *no-lines*属性到。和我们给navbar添加的secondary属性一样，这个属性让我们的列表里面的项不能显示边界。
接下来的有点棘手，也就是我们设置滑动项的地方，和不同的是，我们用的是，由两个部分的内容组成 -- 列表项本身，和，这是是用户滑动列表项显示的。
第一块是普通的定义，但是不是直接使用的我们使用的是实际上是一个用来列表项样式的按钮。视觉上，这两个方法基本上是一样的，但是实际上在移动应用里任何非和元素的点击处理器都会有一点点延迟。我们不喜欢延迟，所以我们用了按钮。
我们给按钮附加的点击处理器调用了一个*viewChecklist*函数同时也传入了一个参数名为*checklist*。我们目前还没有这个函数定义，但是实际上我们会根据一个数据数组创建大量的此类列表项。所以最终此处传入的*checklist*将是点击的想的引用（详情后续讲解）。
最后，我们的第二个代码块，简单的定义了在用户滑动列表项的时候的显示内容。在本案例中我们只是加入了一个‘Edit’和‘Delte’按钮，他们也会在调用函数的时候注入*checklist*引用（再次提醒，我们稍后创建这个原因，目前他会引发问题）。
这就是home页面的所有内容，我们现在去checklist页面。

Checklist页面

跟之前流程一样，在学习之前我们先看看效果图：



看起来跟之前那个差不多，最起码大部分看起来是这样的，但是还是有一些不同点。显然，我们有另外的一个按钮，一个返回按钮用于返回主页。列表项旁边有一个checkbox用于标记一个项是否完成，我们这里也设置了滑动项。

再次，我们将添加模板代码然后来讲解：

> 修改 **src/pages/checklist/checklist.html**为以下：

```
<ion-header>
    <ion-navbar color="secondary">
    <ion-title>
        CHECKLIST TITLE
    </ion-title>
    <ion-buttons end>
    <button ion-button icon-only (click)="uncheckItems()"><ion-icon name="refresh-circ
le"></ion-icon></button>
    <button ion-button icon-only (click)="addItem()"><ion-icon  name="add-circle"></io
n-icon></button>
    </ion-buttons>
    </ion-navbar>
</ion-header>
<ion-content>
<ion-list no-lines>
    <ion-item-sliding>
    <ion-item>
        <ion-label>ITEM TITLE</ion-label>
        <ion-checkbox [checked]="item.checked" (click)="toggleItem(item)">
        </ion-checkbox>
    </ion-item>
    <ion-item-options>
        <button ion-button icon-only color="light" (click)="renameItem(item)"><ion-icon
 name="clipboard"></ion-icon> Edit</button>
        <button ion-button icon-only color="danger"  (click)="removeItem(item)"><ion-i
con name="trash"></ion-icon>
        Delete</button>
    </ion-item-options>
    </ion-item-sliding>
</ion-list>
</ion-content>
```

你已经知道了navbar的如何工作的，本次我们只是在里面新加入了一个按钮，他也使用了**end**属性，这样所有的项都会排列到右边。本次我们将使用的传统方式，显示当前展示的检查列表（最起码，我们马上就会做这个）的标题。按钮也都有不同的点击处理函数，但是由于我们当前是在**ChecklistPage**组件，这些函数都会在**checklist.ts**文件里面触发（这个东西我们当前也没有创建）。

现在你也知道滑动项怎么工作的，但是这次我们有一个作为主体内容，当他被点击的时候他会触发*toggleItem()*函数，这个函数稍后也会定义。注意，我们用来普通的来替代，这是因为我们是直接给checkbox附加点击处理器而不用附加到整个项。

这里也有一点新语法，我们靠近点看：

```
[checked]="item.checked"
```

如果某些东西被 [方括号] 包围的话，因为这我们将在这个元素上修改一个属性**property**，我们将设置引号里面包含的表达式**expression**，而不是字符串。所以，在本案例中，我们将设置**checked**属性到**item.checked**的值。目前我们还没有创建一个**item**的引用所以现在是没有任何作用的，但是后续会。这里需要重点记住的是方括号会评估引号里面的任何内容。我们想象一下你的组件的类里有如下定义：

```
this.myName = "Josh"
```

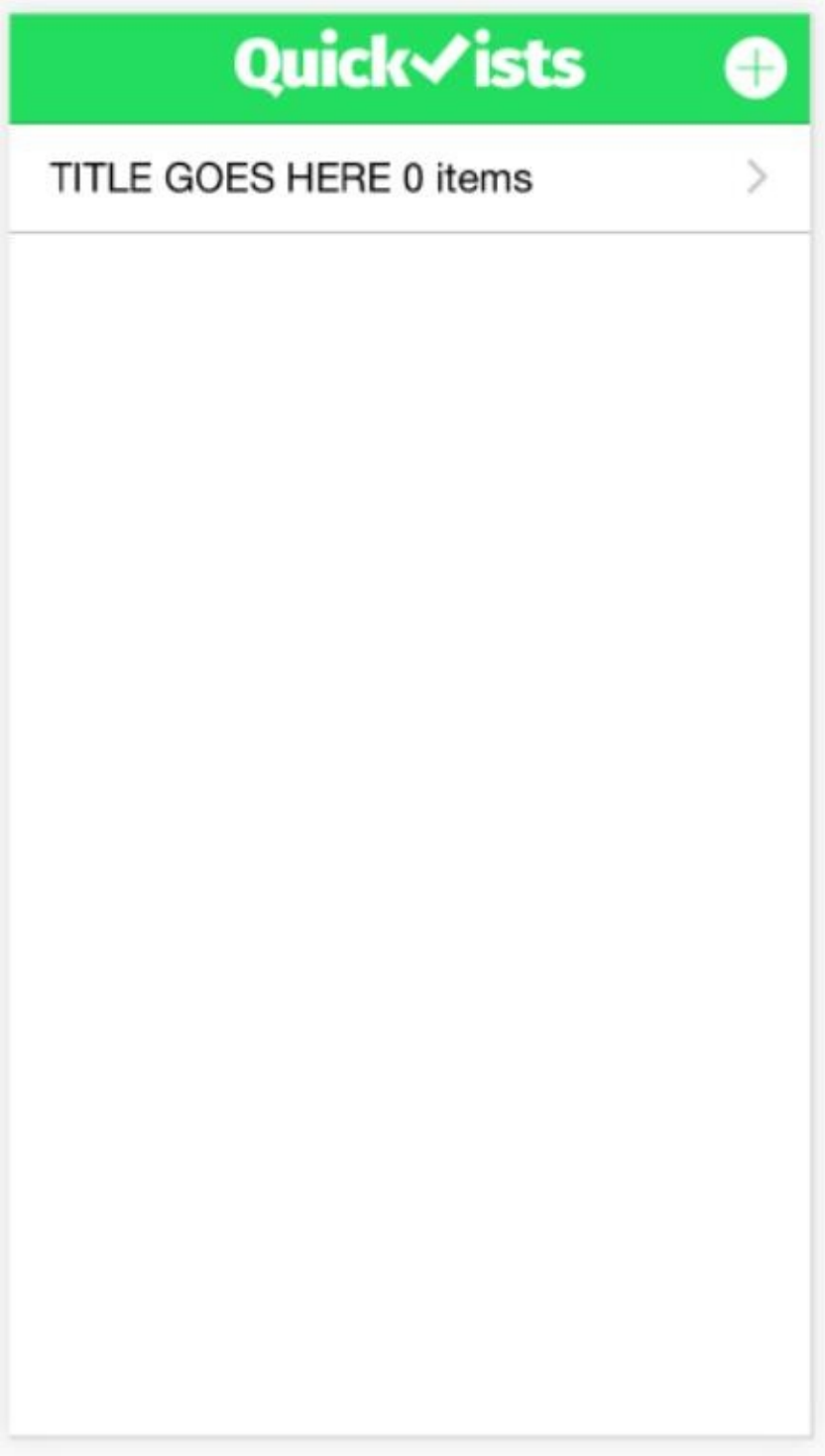
如果我如下使用：

```
<something myName="myName">
```

myName属性**attribute**将被设为“myName”文本值，但是如果我们使用如下代码的话：

```
<something [myName]="myName">
```

myName属性**property**将被设置为‘Josh’，因为myName在此示例中将会评估（译者：表达式估值）。继续，模板里没有别的新玩意了 -- 我们简单的添加了‘Edit’和‘Delete’按钮到滑动列表，就像上一个页面一样。
如果现在运行*ionic serve*，将会看到如下结果：



我想我们都会认为这看起来好丑，但是结构就在那里。请记住，我们还没有对logo指定样式，因此根据你使用何种尺寸的设备，他可能和上图看起来不大一样。
接下来的课程我们将给列表拉取真是数据并对他自定义样式，这样他会看起来顺眼些。Ionic 2的模板语法第一眼看起来可能会有点迷糊，但是一旦你熟悉了之后将会很好使用。

第四课：数据模型和Observable

本节课中我们设计应用中将会用到的检查列表的数据模型，他会和Observable进行合作。数据模型不是Ionic 2特有的东西，模型是变成里面同样概念。根据内容的不同，模型的定义多变，但是总的来说模型就是用于存储和代表数据的。
在Ionic 2和Angular 2中，如果我们想持有某些数据的引用，我们大概需要这样去做：

```
this.myDataArray = ['1', '2', '3'];
```

然后，如果我们是创建模型的话，那么大概是这样子的：

```
this.myDataArray = [
    new MyDataModel('1'),
    new MyDataModel('2'),
    new MyDataModel('3')
];
```

所以，我们创建一个对象**object**来持有数据而不是纯存储。首先，可能比较难理解为什么我们要这么做，对于像上面例子这么简单的数据看起来很复杂，但是同时会带来很多好处。这个应用中带来的主要好处包括：

- 允许我们清晰的定义数据结构
- 允许我们在数据模型里面创建助理函数来操作数据
- 允许我们在不同地方重用数据模型，简化代码

真切的希望本课能清晰的给你展示创建数据模型能够带来的好处，但是我体验说明一下这不是必需的。你也可以直接在类里面定义一些数据就可以了。
我们也在数据模型中会创建和用到我们自己的**Observable**，但是我们还是在遇到他的时候再说吧。

创建数据模型

通常如果我们想要创建一个数据模型的话我们将创建一个类来定义他（基本上就是个普通对象），然后定义一些助理函数，就像这样：

```
class PersonModel {

    constructor(name, age){
        this.name = name;
        this.age = age;
    }

    increaseAge(){
        this.age++;
    }

    changeName(name){
        this.name = name;
    }
}
```

然后，我们就可以从他创建以系列的实例（对象）了：

```
let person1 = new PersonModel('Jason', 43);
let person2 = new PersonModel('Louise', 22);
```

然后实例有可以分别去调用助理函数：

```
person1.increaseAge();
```

Ionic 2的理念也差不多，除了Ionic 2/Angular 2中我们需要创建一个**Injectable**（基础里面讲过）。记住**Injectable**用于创建服务，他创建的服务可以注入到任何的其他组件中，所以当我们想要使用数据模型的时候，我们直接注入进去就可以了。
我们看一下实际的数据模型大概是什么样子的，然后再学习之。
> 修改 **src/models/checklist-model.ts**为以下

```
export class ChecklistModel {
    checklist: any;
    checklistObserver: any;
    constructor(public title: string, public items: any[]){
        this.items = items;
    }

    addItem(item): void {
        this.items.push({
            title: item,
            checked: false
        });
    }

    removeItem(item): void {
        let index = this.items.indexOf(item);
        if(index > -1){
            this.items.splice(index, 1);
        }
    }

    renameItem(item, title): void {
        let index = this.items.indexOf(item);
        if(index > -1){
            this.items[index].title = title;
        }
    }

    setTitle(title): void {
        this.title = title;
    }

    toggleItem(item): void {
        item.checked = !item.checked;
    }
}
```

我们这个数据模型所作的实际上是创建了一个单独的检查列表的蓝图。一个检查列表有一个标题和任意数量的需要完成的项。所以，我们设置了成员变量来持有这些值：title是一个简单的字符串，items是一个数组。

注意，我们允许通过构造器传入标题和项。创建一个新的检查列表的时候必须传入标题，但是项的数组是可选的。如果你想立刻给检查列表添加项的话，我们可以在初始化的时候传入，否则将会初始化一个空的数组。

我们也创建了大量的助理函数，看起来都挺直白的，他们允许我们改变检查列表的标题，或者修改检查列表的项目（修改名字，移除项，给检查列表添加新的项，设置项的完成状态）。

同时主题我们有在每个函数后面加上：*void*。就像我们声明变量的时候指定类型一样：

```
checklist: any
```

我们也可以声明函数的返回值类型。在本案例中，没有返回值，所有我们用的是*void*。如果其中一个函数返回字符串的话，那么我们就用：*string*代替。
设置完以上之后，我们可以在任何导入了Checklist Model（下一课会讲）的组件里简单的通过以下代码创建一个新的检查列表：

```
let newChecklist = new ChecklistModel('My Checklist', []);
```

或者

```
let newChecklist = new ChecklistModel('My Checklist', myItemsArray);
```

我们现在将做些更好玩的，在数据模型中使用**Observable**，这样我们就可以在检查列表修改的时候被告知（此时我们可以触发保存数据到内存）。

添加一个Observable

在本课程的基础部分我们有了解一点Observable -- 我们使用**Http**服务返回的Observable来刷新一下你的记忆：

```
this.http.get('https://www.reddit.com/r/gifs/new/.json?limit=10').map(res => res.json(
)).subscribe(data => {
    console.log(data);
});
```

我们调用了*get*方法，然后订阅*subscribe*他返回的**Observable**。记住，Observable和Promise不一样，他返回的是数据流，并且可以多次获得数据。这个理念无法通过**Http**服务来阐述，因为大部分情况下我们只需要获取一次数据。在Http案例中Observable也已经为我们建好了。
我们也将在数据模型中从头到尾的创建我们自己的Observable，这样当检查列表发生改变的时候（因为我们在发生改变的时候会发出一些数据）我们可以在应用的其他部分可以监听到。在实施Observable的时候，你可以看到如何从头创建一个observable，你也可以看到一个Observable如何多次发出数据。
实施之前，我们更详细的讲一下Observable，也就是我们先要具体要做的内容。在以上的subscribe函数代码中，我们只处理了一次response：

```
this.http.get(url).subscribe(data => {
    console.log(data);
});
```

这实际上就是Observable的 onNext 响应。Observable同时也提供了两个其他的响应，onError和onCompleted，如果我们愿意的话可以处理全部：

```
this.http.get(url).subscribe(
    (data) => {
        console.log(data);
    },
    (err) => {
        console.log(err);
    },
    () => {
        console.log("completed");
    }
);
```

以上代码第一个事件处理器处理的是 onNext 想要，基本意思就是“当我们侦测到从流里传来下一点数据的时候，那么就调用这里”。第二个处理器处理的是 onError 想要，你可能猜到了他是当有错误发生的时候就会触发。最后的处理器处理的是 onCompleted 事件，他是当Observable返回所有数据后触发的。
这里最有用的是 onNext了，如果我们创建自己的observable，我们可以通过调用Observable的 *next* 提供一些数据多次触发 onNext 响应。
现在，理论的道路清理完毕，我们看一下具体实现。
> 修改 **src/models/checklist-model.ts**为如下

```
import {Observable} from 'rxjs/Observable';

export class ChecklistModel {
    checklist: any;
    checklistObserver: any;
    constructor(public title: string, public items: any[]){
        this.items = items;
        this.checklist = Observable.create(observer => {
            this.checklistObserver = observer;
        });
    }

    addItem(item): void {
        this.items.push({
            title: item,
            checked: false
        });
        this.checklistObserver.next(true);
    }

    removeItem(item): void {
        let index = this.items.indexOf(item);
        if(index > -1){
            this.items.splice(index, 1);
        }
        this.checklistObserver.next(true);
    }

    renameItem(item, title): void {
        let index = this.items.indexOf(item);
        if(index > -1){
            this.items[index].title = title;
        }
        this.checklistObserver.next(true);
    }

    setTitle(title): void {
        this.title = title;
        this.checklistObserver.next(true);
    }

    toggleItem(item): void {
        item.checked = !item.checked;
        this.checklistObserver.next(true);
    }
}
```

此处第一件要关心的事情是我们从RxJS库里导入了**Observable**。然后在构造器中，我们设置了Observable：

```
this.checklist = Observable.create(observer => {
    this.checklistObserver = observer;
});
```

代码里面的*this.checklist*成员变量是我们自己的observable。由于他是一个observable，我们可以对他进行订阅，由于他是我们数据模型的一部分，我们可以在任何检查列表上订阅他。例如：

```
let newChecklist = new ChecklistModel('My Checklist', []);

newChecklist.checklist.subscribe(data => {
    console.log(data);
});
```

当然，我们还没有用Observable来做什么，所以他永远都不会触发*onNext*响应。这就是为什么我们在每个助理函数里面添加如下代码片的原因：

```
this.checklistObserver.next(true);
```

这样，无论何时我们通过助理函数来更改标题，或者添加一个新的项，或者其他任何事情的时候，他都会通知所有订阅了他的Observable的任何事物。我们只要知道发生了改变所以我们只要传入一个boolean（true或者false），但是如果我们想要的话我们也可以传入一些数据。
这样的结果就是现在我们可以“观察observe”我们创建的检查列表的变动。稍后我们将用到这些变更监听然后触发保存操作。

总结

本课程中我们稍微超出了入门者级别，创建了强壮的（robust，据说有人翻译为鲁棒性）数据模型。如我之前讲到，他当然有他的好处，但是如果学习本课程遇到麻烦的时候也不要被吓到 -- 作为初学者的你可以直接在类上定义数据，不要操心数据模型和observable。
我真心不想用Observable吓坏你 -- 他们很容易混淆（直到你脑子里面适应他们），在除了Http服务订阅响应之外，在大部分的应用中你真心不需要用到它们。但是，一旦你理解了他们，你就可以用他们来做更强大的东西。
尽管本课程稍微高级一点，他还是很好的向你展示了如何在项目中使用Observable，如果想跟上课程进度，希望接下来的内容不会烦到你。

第五课：创建checklist与列表项

目前为止我们做了很多的设置和架构工作，但是本课中我们开始制作骨架部分。我们将加入创建checklist，查看checklist以及给他们添加项（同时也提供编辑项和列表的操作）的途径。这将是一个大动作，建议你先准备点咖啡。

checklist

第一件要做的事情是添加创建和展示checklist所需的所有东西。意思是添加到类定义，修改之前创建的模板来展示真实的清单数据。
先从设置类定义开始。
> 修改 **src/pages/home/home.ts**为如下：

```
import { Component } from '@angular/core';
import { NavController, AlertController, Platform } from 'ionic-angular';
import { ChecklistPage } from '../checklist/checklist';
import { ChecklistModel } from '../../models/checklist-model';
import { Data } from '../../providers/data';
import { Keyboard } from 'ionic-native';

@Component({
    selector: 'page-home',
    templateUrl: 'home.html'
})
export class HomePage {
    checklists: ChecklistModel[] = [];
    constructor(public nav: NavController, public dataService: Data, public  alertCtrl
: AlertController, public platform: Platform) {
    }

    ionViewDidLoad() {
    }

    addChecklist(): void {
    }

    renameChecklist(checklist): void {
    }

    viewChecklist(checklist): void {
    }

    removeChecklist(checklist): void{
    }

    save(): void{
    }
}
```

在这里我们从Ionic库里面导入了不少东西。**NavController**你可以已经很熟悉了，但是**AlertController**可能就不那么熟悉了。AlertController允许我们向用户展示各种警告框，包括基本的提示框，带输入的提示框，确认框等等。我们以他作为添加新checklist的途径。
同时我们也导入了我们的**ChecklistPage**，我们稍后会实现他，最重要的是我们导入了上节课制作的Checklist Model。
我们也导入了之前生成的Data提供者，但是他的具体功能后续才会实现。最后，我们从Ionic Native导入了'Keyboard'，这样我们可以用他来确保稍后用户添加了checklist之后能够关闭软键盘。
在构造器中通过给NavController和Data添加*public*修饰符，我们就可以在类里通过*this.nav*和*this.dataService*来访问他们了。基本上他是以下内容的简短写法：

```
constructor(nav: NavController, dataService: Data){
    this.nav = nav;
    this.dataService = dataService;
}
```

之前你应该看到过。
我们也声明了一个*checklists*数组，类定义里面可以通过*this.checklists*来访问他。类定义里面的大量的函数将在后续一个一个的讲解。

addChecklist

这个方法提供给用户创建一个新的checklist。他会启动一个提示框，然后使用其中输入的信息来创建一个新的checklist（会用到我们之前创建的数据模型）。
> 修改 **addChecklist** 函数如下：

```
addChecklist(): void {
    let prompt = this.alertCtrl.create({
        title: 'New Checklist',
        message: 'Enter the name of your new checklist below:',
        inputs: [
            {
                name: 'name'
            }
        ],
        buttons: [
            {
                text: 'Cancel'
            },
            {
                text: 'Save',
                handler: data => {
                    let newChecklist = new ChecklistModel(data.name, []);
                    this.checklists.push(newChecklist);

                    newChecklist.checklist.subscribe(update => {
                        this.save();
                    });

                    this.save();
                }
            }
        ]
    });

    prompt.present();
}
```

我们给用户展示了一个提示框，提示框包含了一个输入域，两个按钮Cancel和Save。取消按钮除了关闭提示框之外，不做其他事情，但是保存按钮我们给他添加了一个处理器用于传递输入的数据。
在处理器里面，我们先通过传入输入的名字到一个checklist model里生成了一个新的checklist，然后将他**push**到咱们的*this.checklists*数组。然后我们订阅了我们给数据模型添加的observable来监听checklist的修改，然后其中会触发save函数。注意，我们这里调用了两次save，一个是observable触发的，另一个是直接触发的（因为我们添加了一个新的checklist）。
最后，我们通过*present*方法呈现了这个提示框。
如果你再次看模板文件的时候，会发现我们在add按钮点击的时候已经调用了这个函数。

```
<button (click)="addChecklist()"><ion-icon name="add-circle"></ion-icon></button>
```

renameChecklist

接下来我们来定义*renameChecklist*函数，看名字就知道是给checklist重命名的。
> 修改 **renameChecklist** 函数如下：

```
renameChecklist(checklist): void {
    let prompt = this.alertCtrl.create({
        title: 'Rename Checklist',
        message: 'Enter the new name of this checklist below:',
        inputs: [
            {
                name: 'name'
            }
        ],
        buttons: [
            {
                text: 'Cancel'
            },
            {
                text: 'Save',
                handler: data => {
                    let index = this.checklists.indexOf(checklist);
                    if(index > -1){
                        this.checklists[index].setTitle(data.name);
                        this.save();
                    }
                }
            }
        ]
    });

    prompt.present();
}
```

第一眼看起来和*addChecklist*函数很想，因为他本来就很像哇。用了同样的提示框，同样的输入框，童颜的按钮，只是处理器稍有区别而已。
注意，这个函数有参数传入，也就是我们将要改名的checklist的引用。我们稍后修改模板来传入此参数，但是此刻我们假装他已经传入进来就可以了。
我们使用此checklist的引用在*this.checklists*中查找然后将他设置为新的标题，然后触发save。
你可能会记得我们已经在模板中设置了一个点击处理器来调用这个函数：

```
<button light (click)="renameChecklist(checklist)"><ion-icon name="clipboard"></ion-ic
on></button>
```

removeChecklist

接下来我们来加入删除checklist的能力。
> 修改 **removeChecklist** 函数如下：

```
removeChecklist(checklist): void{

    let index = this.checklists.indexOf(checklist);

    if(index > -1){
        this.checklists.splice(index, 1);
        this.save();
    }

}
```

这个函数简单多了，因为他没有用户输入需求，我们只需要处理掉这个checklist就可以了。跟之前做的一样，我们传入了一个checklist引用然后在*this.checklists*找到他。之后通过数组的**splice**方法简单的将他移除掉然后触发save就完成了。
以下是模板中触发此函数的相关代码：

```
<button danger (click)="removeChecklist(checklist)"><ion-icon name="trash"></ion-icon>
 Delete</button>
```

viewChecklist

我们现在可以创建和修改咱们的checklist了，但是我们也得可以看checklist的详情不是。我们将要通过咱们的NavController来**push**压入一个新页面然后传入我们选择的checklist。
> 修改 **viewChecklist** 函数为如下：

```
viewChecklist(checklist): void {
    this.nav.push(ChecklistPage, {
        checklist: checklist
    });
}
```

我们将之前导入的**ChecklistPage**（还没完成）传入到 push 方法，以及想要传递给页面的其他参数，也就是我们要显示详情的checklist。在页面中，我们就可以通过**NavParams**来获取这些数据了。

save

这可能是现在来讲最奇怪的一个方法了，我们实际现在不会去实现。后续的保存和加载数据才是他的本命课程，我们那个时候才会再来看他。

为了把所有事情联合起来，我们需要完善home页面的模板。我们现在可以用自己的数据了，但是我们还是什么都看不到。
> 修改 **src/pages/home/home.html** 为如下：

```
<ion-header>
    <ion-navbar color="secondary">
        <ion-title>
            <img src = "assets/images/logo.png" />
        </ion-title>
        <ion-buttons end>
            <button ion-button icon-only (click)="addChecklist()"><ion-icon  name="add
-circle"></ion-icon></button>
        </ion-buttons>
    </ion-navbar>
</ion-header>

<ion-content>
    <ion-list no-lines>
        <ion-item-sliding *ngFor="let checklist of checklists">
        <button ion-item (click)="viewChecklist(checklist)">
            {{checklist.title}}
            <span>{{checklist.items.length}} items</span>
        </button>
        <ion-item-options>
            <button ion-button icon-only color="light" (click)="renameChecklist(checkl
ist)"><ion-icon
            name="clipboard"></ion-icon></button>
            <button ion-button icon-only color="danger" (click)="removeChecklist(check
list)"><ion-icon
            name="trash"></ion-icon></button>
        </ion-item-options>
        </ion-item-sliding>
    </ion-list>
</ion-content>
```

这里我们做了很多有趣的事情，但是最需要注意的是 ngFor 循环：

```
<ion-item-sliding *ngFor="let checklist of checklists">
```

这个循环的作用是循环我们的*this.checklist*数组并为他们在列表中创建一个滑动项。记住，在ngFor前面使用的 * 语法是Angular 2中用于创建内置模板的快捷方式，所以我们实际上是为数组中的每个元素创建一个模板。每次新建模板的时候他都会包含指定项的信息，这样在 ngFor 里面的任意处我们都可以获取指定checklist的数据用作渲染，如下：

```
{{checklist.title}}
```

同时也请注意 ngFor 循环的 *checklist* 变量前面都有 *let* 修饰符。在Angular 2中使用*let*修饰符让我们可以创建一个局部变量，这样我们可以将这个局部变量的引用传入到之前创建的函数中。为了便于理解，如果我们用以下方法替换的话：

```
<ion-item-sliding *ngFor="let check of checklists">
```

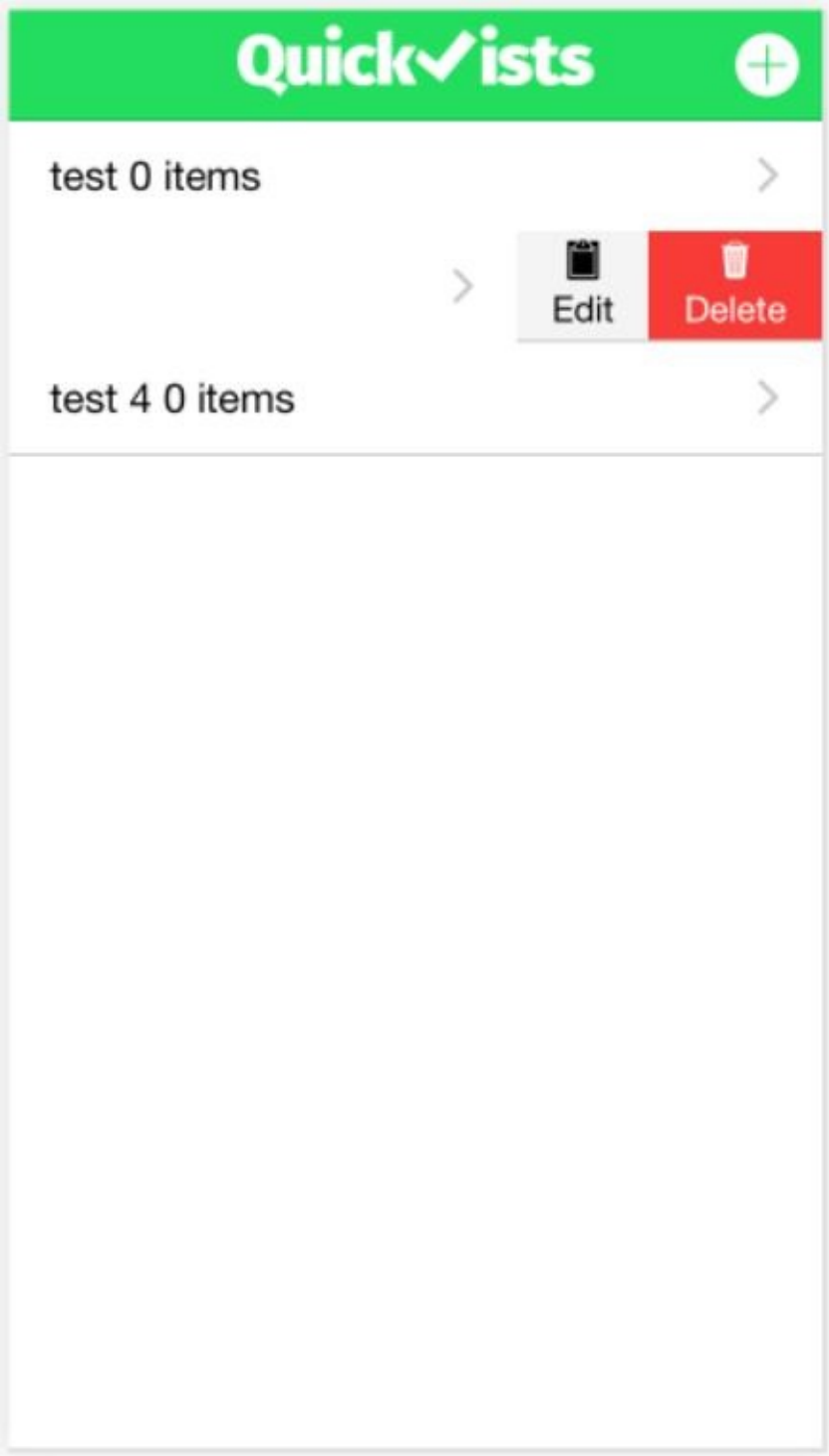
我们就可以这样去渲染数据：

```
{{check.title}}
```

可以这样穿个函数：

```
removeChecklist(check)
```

这样我们算是完成了home页面了，现在你可以去添加，编辑和删除checklist了，也可以显示指定checklist（现在没有任何实际内容，所以算是用不了）的详情页了。
此时当你运行*ionic serve*的时候，你应该可以看到这样的画面：



重点注意如果想要创建一个新的项然后展示他的详情页的话，必须从**checklist.html**移除此行：

```
<ion-checkbox [checked]="item.checked" (click)="toggleItem(item)"></ion-checkbox>
```

后续课程里面我们只会在有项数据的时候才加入他，但是目前为止，由于没有数据所以在尝试访问数据的时候会报错。
所有这些后面都会讲到，如果你现在想正常预览应用的话得先做这些额外的步骤。

Checklist 项

现在我们可以触发checklist详情页了，我们最好加入点内容，给用户提供创建和修改单个checklist数据项的途径。这个部分中，我们将实现自己的Checklist Page，它将通过home页面启动，会提供一些相关数据给他用于展示checklist。
我们从设置类定义开始。
> 修改 **src/pages/checklist/checklist.ts** 为如下：

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

@Component({
    selector: 'page-checklist',
    templateUrl: 'checklist.html'
})

export class ChecklistPage {
    checklist: any;
    constructor(public nav: NavController, public navParams: NavParams, public
    alertCtrl: AlertController){
        this.checklist = this.navParams.get('checklist');
    }

    addItem(): void {
    }

    toggleItem(item): void {
    }

    removeItem(item): void {
    }

    renameItem(item): void {
    }

    uncheckItems(): void {
    }

}
```

这里需要学习的点不多，唯一需要注意的是**NavParams**的使用。当我们传递数据到另一个页面的时候，我们可以通过注入NavParams然后通过他的get方法来获取。在本例中，我们只是传入了想要查看的checklist，如果你想要的话可以传入很多其他你需要的值。
跟之前一样，我们将一个一个的去实现每个方法。很多跟之前home页面方法相同。

addItem

> 修改 **addItem** 函数为如下：

```
addItem(): void {
    let prompt = this.alertCtrl.create({
        title: 'Add Item',
        message: 'Enter the name of the task for this checklist below:',
        inputs: [
            {
                name: 'name'
            }
        ],
        buttons: [
            {
                text: 'Cancel'
            },
            {
                text: 'Save',
                handler: data => {
                    this.checklist.addItem(data.name);
                }
            }
        ]
    });

    prompt.present();
}
```

这些看起来应该很熟悉，但是注意处理器的不同。由于我们在数据模型中创建了一个**addItem**助理函数，我们只需要调用这个函数传入我们需要创建的项的名即可（告诉过你数据模型会带来很大的便利！）。

renameItem

> 修改 **renameItem** 函数为如下：

```
renameItem(item): void {
    let prompt = this.alertCtrl.create({
        title: 'Rename Item',
        message: 'Enter the new name of the task for this checklist below:',
        inputs: [
            {
                name: 'name'
            }
        ],
        buttons: [
            {
                text: 'Cancel'
            },
            {
                text: 'Save',
                handler: data => {
                    this.checklist.renameItem(item, data.name);
                }
            }
        ]
    });

    prompt.present();
}
```

基本上相同除了在处理器中调用了数据模型的**renameItem**方法，同时传入了需要改名的项的引用。

removeItem

> 修改 **removeItem** 函数为如下：

```
removeItem(item): void {
    this.checklist.removeItem(item);
}
```

这个更简单，我们简单的调用了数据模型的*removeItem*助理函数然后传入了我们需要删掉的项。

toggleItem

> 修改 **toggleItem** 函数为如下：

```
toggleItem(item): void {
    this.checklist.toggleItem(item);
}
```

这个函数用于切换单独项的标记为开或者关，我们也只是简单的传入项的引用给数据模型。

uncheckItems

> 修改 **uncheckItems** 函数为如下：

```
uncheckItems(): void {
    this.checklist.items.forEach((item) => {
        if(item.checked){
            this.checklist.toggleItem(item);
        }
    });
}
```

这个函数是绑定到模板里的reset按钮的，他将循环checklist里面的每个项，如果当前项是开的状态的话，调用数据模型的*toggleItem*方法关掉他。这个方法让用户可以一次关掉所有项。现在，我们只剩下checklist页需要处理了。我们之前已经设置好了这个模板的整体结构，但是为了显示真实数据，我们也需要对他进行home页类似的调整。

> 修改 **src/pages/checklist/checklist.html** 如下：

```
<ion-header>
    <ion-navbar color="secondary">
        <ion-title>
        {{checklist.title}}
        </ion-title>
        <ion-buttons end>
            <button ion-button icon-only (click)="uncheckItems()"><ion-icon    name="r
efresh-circle"></ion-icon></button>
            <button ion-button icon-only (click)="addItem()"><ion-icon    name="add-ci
rcle"></ion-icon></button>
        </ion-buttons>
    </ion-navbar>
</ion-header>
<ion-content>
    <ion-list no-lines>
        <ion-item-sliding *ngFor="let item of checklist.items">
            <ion-item>
                <ion-label>{{item.title}}</ion-label>
                <ion-checkbox [checked]="item.checked" (click)="toggleItem(item)">
                </ion-checkbox>
            </ion-item>
        <ion-item-options>
        <button ion-button icon-only color="light" (click)="renameItem(item)"><ion-icon
 name="clipboard"></ion-icon></button>
        <button ion-button icon-only color="danger" (click)="removeItem(item)"><ion-ic
on name="trash"></ion-icon></button>
        </ion-item-options>
        </ion-item-sliding>
    </ion-list>
</ion-content>
```

看起来跟home页面很像，但是有些许不同。我们用类定义中的checklist数据在navbar中显示当前checklist的标题。我们再次使用 ngFor 来循环数据，但是这次我们只是循环的数据项，也就是checklist的子项。同时注意我们使用了双花括号来渲染数据：

```
{{item.title}}
```

我们也可以通过方括号来设置元素属性：

```
[checked]="item.checked"
```

他将会把**checked**的值得设置为*item.checked*的值。

总结

现在你基本可以执行应用里的每个函数来，包括创建checklist，修改，查看单独的checklist，以及给他们添加单独的数据项。
试试看在浏览器中运行你的应用，添加你自己的checklist和数据项。
下一节课中，我们将实现保存数据然后看看怎么样美化这个原因（先屋子再门面，对吧？）。

第六课：保存和加载数据

你知道什么是最烦恼的吗？如果你创建了一大个待完成的checklist，稍后回来的时候再用的时候，发现没有了。嗯，这就是应用当前的状态，所以我们将要给用户提供一个保存用户数据的途径。
我们已经为他做了一些架构准备，我们订阅了我们的Observable在每次数据变更的时候都调用了*save*函数，我们只需要现在来实现这个函数就可以了。
> 修改 **src/pages/home/home.ts** 里面的 **save** 方法：

```
save(): void {
    Keyboard.close();
    this.dataService.save(this.checklists);
}
```

如果你不健忘的话，早些时候我们已经生成和导入了一个数据服务，所以我们这里只需要调用他并传入checklists数据。当然，我们目前还没有实现这个数据服务，所以他现在不会对数据做任何事情，我们现在就来弥补他。注意，我们也调用了Keyboard的close方法，由于会在任何数据项变更的时候调用*save*方法，我们可以用他来在用户添加或者编辑完数据之后确保键盘关闭。

保存数据

我们现在要给**data.ts**码代码了，让他可以将传入的任何数据保存到存储里面去。实际上这个服务的代码相当的简单，我们来看看： **>** 修改 **src/providers/data.ts** ：

```
import { Storage } from '@ionic/storage';
import { Injectable } from '@angular/core';

@Injectable()
export class Data {
    constructor(public storage: Storage){
    }

    getData(): Promise<any> {
        return this.storage.get('checklists');
    }
    save(data): void {
        let saveData = [];
        //Remove observables
        data.forEach((checklist) => {
            saveData.push({
                title: checklist.title,
                items: checklist.items
            });
        });

        let newData = JSON.stringify(saveData);
        this.storage.set('checklists', newData);
    }
}
```

这里有个我们之前没见过的导入：

```
import { Storage } from '@ionic/storage';
```

Storage是Ionic的统一存储服务，他负责使用最好的方式来存储数据的同时提供了一个统一的API给供我们使用。
当在设备上运行的时候，如果SQLite插件是可用状态的话（我们之前安装过了），他将会使用本地（native）SQLite数据库来存储数据。由于SQLite数据库只在设备上可用，在SQLite不可用的情况下，Storage也会用IndexedDB，WebSQL或者标准浏览器的localStorage。
尽可能使用SQLite，因为基于浏览器的本地存储不大可靠可以被操作系统默默的清理掉。如果你的数据可以被随机清理掉的话显然很不理想。
我们来看看*getData*函数：

```
getData(): Promise<any> {
    return this.storage.get('checklists');
}
```

这个函数允许我们获取最新的存储数据，他会以**Promise**的方式返回数据。我们将此函数的返回的Promise的返回类型设置为类型，这是一个更复杂的类型。记住，这不是唯一可用类型，所以，如果你觉得很迷惑的话，你可以什么都不加，像这样：

```
getData(){
    return this.storage.get('checklists');
}
```

注意，在这里我们没有给promise完成的时候添加处理器，我们只是返回get方法的结果（一个解析当前存储数据的promise）。记住，这个行为不是瞬间完成的，这样我们可以在调用这个函数的任何地方添加处理器，也更像应用的工作流。（希望这个能够更简单明了）
然后，我们来到了*saveData*函数，这个函数用于将数据保存到存储里：

```
save(data): void {
    let saveData = [];

    //Remove observables
    data.forEach((checklist) => {
        saveData.push({
            title: checklist.title,
            items: checklist.items
        });
    });

    let newData = JSON.stringify(saveData);
    this.storage.set('checklists', newData);
}
```

之前提到我们，我们将数据存为一个简单的JSON编码字符串，所以我们调用了*JSON.stringify*函数然后使用*set*方法将他保存到存储对象里去。在做这些之前，我们把数据的observable移除掉只加入title和items，因为他们跟JSON不搭调（他会引起circular对象错误），我们后面会重新创建它们。
这就是保存数据的全部内容，也没那么复杂。我们接下来处理加载数据到应用中。

加载数据

任何时候当用户打开应用，我们都需要从存储中加载checklists数据，所以最适合做这个操作的地方是home页的*ionViewDidLoad*，这个函数会在页面加载完成的时候触发。我们将对构造器进行小小的变更。
> 在 **src/pages/home/home.ts** 中修改**constructor**和**ionViewDidLoad**：

```
constructor(public nav: NavController, public dataService: Data, public
alertCtrl: AlertController, public platform: Platform) {
}
ionViewDidLoad(){
this.platform.ready().then(() => {
    this.dataService.getData().then((checklists) => {

        let savedChecklists: any = false;

        if(typeof(checklists) != "undefined"){
            savedChecklists = JSON.parse(checklists);
        }
        if(savedChecklists){
            savedChecklists.forEach((savedChecklist) => {
                let loadChecklist = new ChecklistModel(savedChecklist.title, savedChec
klist.items);
                this.checklists.push(loadChecklist);
                loadChecklist.checklist.subscribe(update => {
                    this.save();
                });
            });
        }
    });
});
}
```

我们调用了数据服务里面刚定义的*getData*函数。之前也提过，getData函数返回一个promise而不是直接返回数据，这样我们可以在加载完成之后处理响应。如果getData直接返回数据的话，当我们想要访问他的时候可能他都没有返回。

所以我们一直等待取回数据，然后将checklists传入到我们的处理器。首先我们解码JSON字符串到数组到这样我们就可以用来，然后我们循环数组里的每个数据项，基于这些数据创建一个新的Checklist Data。循环数据和创建新的模型而不是直接将*this.checklists*设置为*savedChecklists*的原因是在保存checklists将他转换成JSON字符串的时候他就丧失了使用数据模型里的助理函数的能力。所以，我们得利用取得的标题和数据项来为所有的checklists创建新的对象。

最后，我们重新给Observable设置监听器，这样当数据改变的时候就会被处罚。重点关注我们所作的这些都是在*platform.ready()*调用里面进行的，如果在设备准备好之前这么做的话将会发生问题。

总结

这就是本节课的所有内容，数据现在在进行任何变更后可以被保存到SQLite数据里面去，应用重新打开的时候，所有数据将会重新加载回来。试一下添加一些checklist或者修改checklist的状态，然后重新加载应用看看对他进行的变更是否还在。

第七课：制作引导滑页与定制主题

这是本应用的最后一节课，我们将添加一些最终的方案来优化用户体验。我们将添加一个滑页手册来向用户展示如何使用本应用（只会在第一次打开应用的时候展示），然后添加一些样式让应用看起来漂亮一些。
我们先来看看滑页。

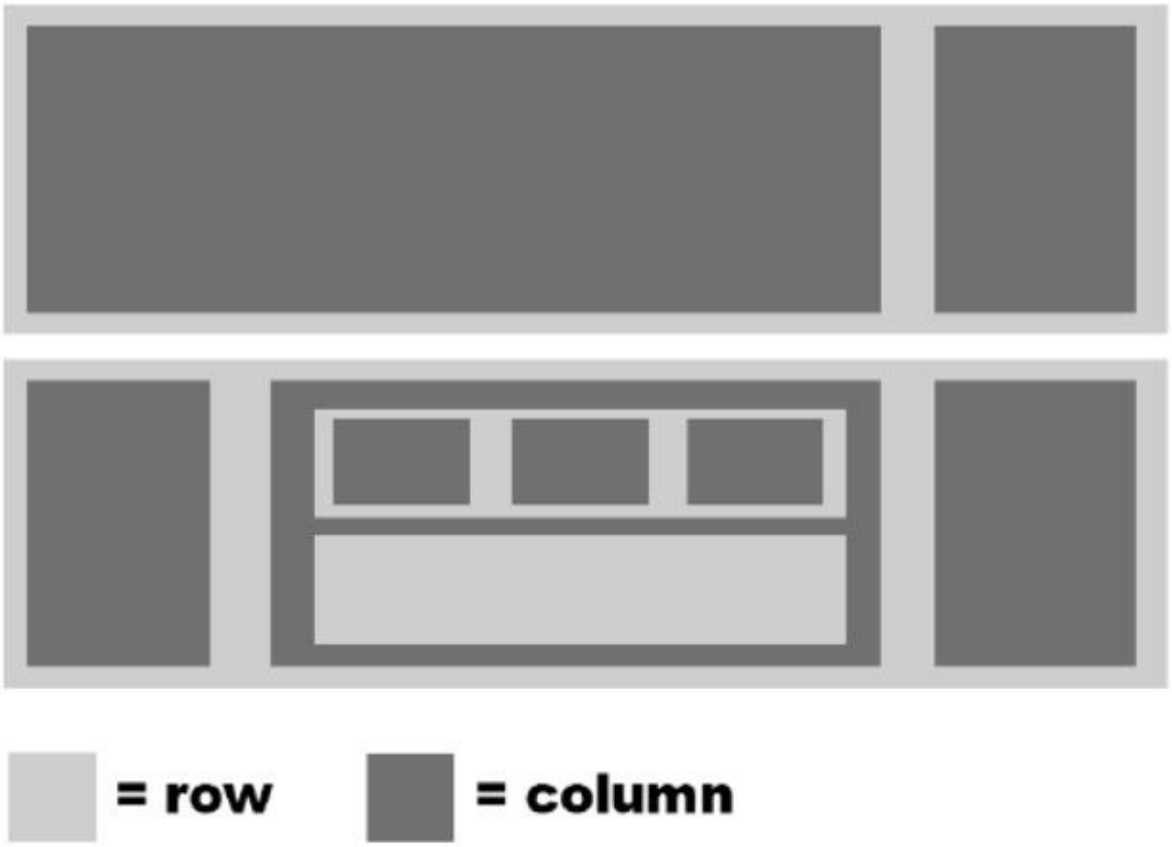
滑动组件

使用滑动卡片手册来向用户做应用介绍说非常通用的一个做法。Ionice有一个内置的滑动组件，所以我们将用它来实现，然后确保用户不用每次打开应用都看到他。
滑动组件本身就很简单，可以用来展示一系列的图片，最后一个滑动页里面是一个按钮用于进入应用。
首先，我们创建好滑动组件，然后我们研究如何将他整合到我们的应用中。我们从创建模板开始。
> 修改**src/pages/intro/intro.html** 如下：

```
<ion-content>
    <ion-slides [options]="slideOptions">
        <ion-slide>
            <img src="assets/images/slide1.png" />
        </ion-slide>

        <ion-slide>
            <img src="assets/images/slide2.png" />
        </ion-slide>

        <ion-slide>
            <img src="assets/images/slide3.png" />
        </ion-slide>

        <ion-slide>
            <ion-row>
                <ion-col>
                    <button ion-button color="light" (click)="goToHome()" style="margi
n-top:20px;">Start Using Quicklists</button>
                </ion-col>
            </ion-row>

            <ion-row>
                <ion-col>
                    <img src="assets/images/slide4.png" />
                </ion-col>
            </ion-row>
        </ion-slide>
    </ion-slides>
</ion-content>
```

第一眼你就会发现模板里并没有navbar，只有内容区域。并不是所有的页面都需要包含导航条的，我们这里就不需要。余下的代码就非常简单了，我们用到了一个带有**options**属性的，这样我们稍后可以在类定义中给他配置一些选项，每个滑动页都是以来定义的。
这样一来，以上代码中用户首先会看到一个容纳了**slide1**图片的滑动也，然后放他们滑动的时候将火看到**slide2**等等知道他们到最后一个页面，上面有一个按钮直达home页。
最后一个滑动也稍微复杂些，因为我们用到了和这样我们可以随心所欲的摆放按钮。这两个

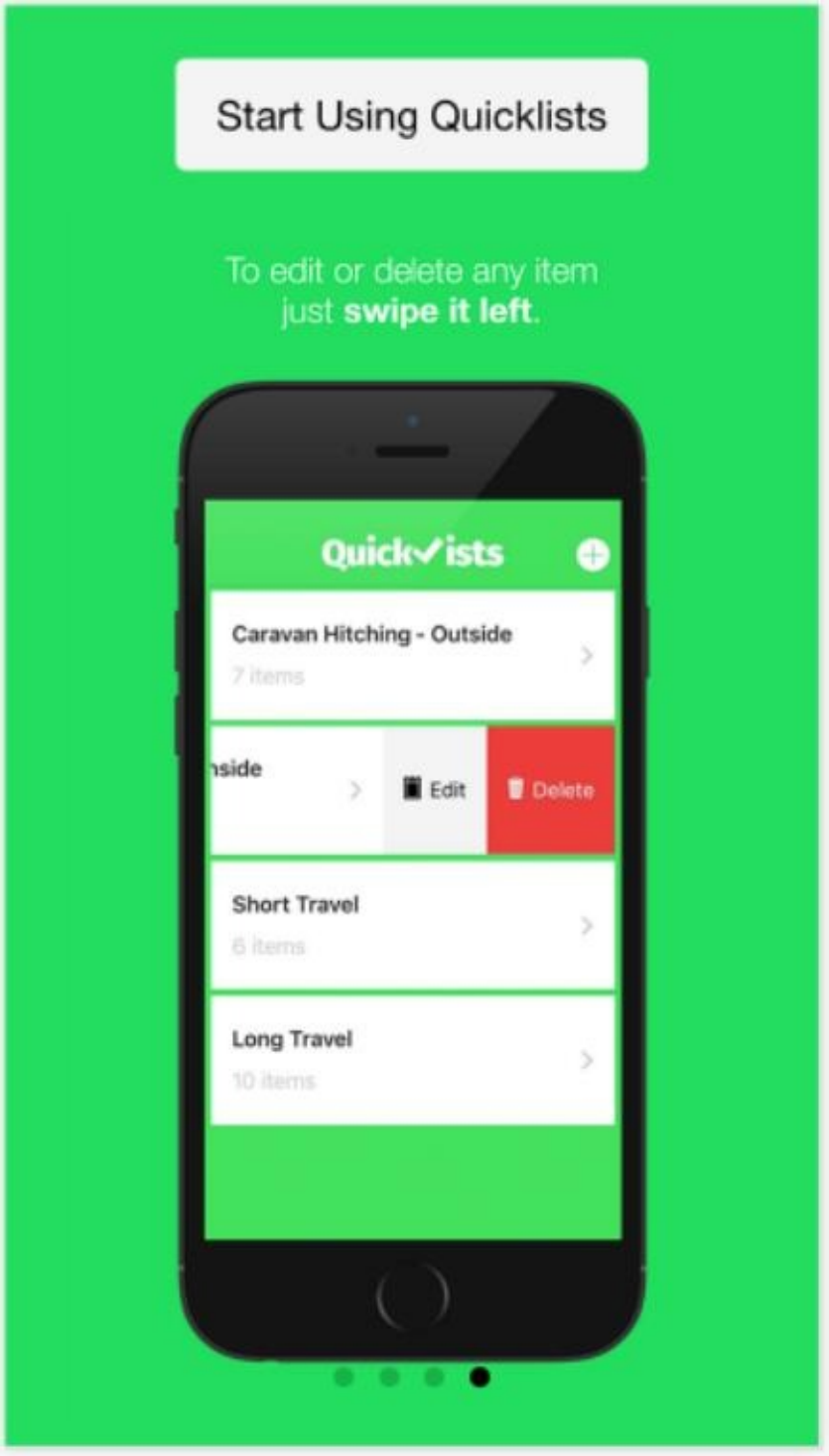
指令组成了Ionic 2的grid栅格系统，row是一个一个堆起来的，col是边靠边列出来的。一下图标可以清晰的表达这一点：



这是个比较简单的范例，我们只是想让按钮显示到图片的顶部，但是你也可以通过给列提供宽度来制作更复杂的布局：

```
<ion-row>
    <ion-col width-10></ion-col>
    <ion-col width-50></ion-col>
</ion-row>
```

你可以嵌套任意你想要的层数。咱们的栅格布局效果图是这样的：



现在我们来定义intro组件的类。

> 修改 **src/pages/intro/intro.ts**为如下：

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

@Component({
    selector: 'page-intro',
    templateUrl: 'intro.html'
})

export class IntroPage {

    slideOptions: any;
    constructor(public nav: NavController){
        this.slideOptions = {
            pager: true
        };
    }

    goToHome(): void {
        this.nav.setRoot(HomePage);
    }
}
```

这是不是迄今为止见过的最简单的类？他所作的知识导入home页面，给我们的slider设置pager属性，当调用*goToHome*的时候通过NavController改变根页面。这样我们在最后一个滑动页点击按钮的时候就可以跳转到home页面，但是我们可能会遇到点问题。当用户每次打开这个原因的时候都要看一遍这个用户手册。要解决这个问题的时候，我们需要对**home.ts**做一个改动。

> 修改 **src/pages/home/home.ts**为如下：

```
import { Component } from '@angular/core';
import { NavController, AlertController, Platform } from 'ionic-angular';
import { ChecklistPage } from '../checklist/checklist';
import { ChecklistModel } from '../../models/checklist-model';
import { Data } from '../../providers/data';
import { Keyboard } from 'ionic-native';
import { IntroPage } from '../intro/intro';
import { Storage } from '@ionic/storage';
```

> 修改 **src/pages/home/home.ts** 的**ionViewDidLoad**方法如下：

```
checklists: ChecklistModel[] = [];

constructor(public nav: NavController, public dataService: Data, public alertCtrl: Ale
rtController, public storage: Storage, public platform:Platform) {

}

ionViewDidLoad(){

    this.platform.ready().then(() => {
        this.storage.get('introShown').then((result) => {
            if(!result){
                this.storage.set('introShown', true);
                this.nav.setRoot(IntroPage);
            }
        });

        this.dataService.getData().then((checklists) => {
            let savedChecklists: any = false;
            if(typeof(checklists) != "undefined"){
                savedChecklists = JSON.parse(checklists);
            }

            if(savedChecklists){
                savedChecklists.forEach((savedChecklist) => {

                    let loadChecklist = new ChecklistModel(savedChecklist.title, saved
Checklist.items);

                    this.checklists.push(loadChecklist);

                    loadChecklist.checklist.subscribe(update => {
                        this.save();
                    });
                });
            }
        });
    });
}
```

我们再次导入和用到了Storage。我们会用他来存储一个标记告诉我们是否有展示过用户手册。
这样我们设置好了新的存储，检查*introShown*标记的存在。如果不存在的话我们将转到介绍页面然后将标记值设为true这样下次他就不会展示了。
注意：测试的时候，如果你想清除此标记值，那么直需要删除浏览器创建的WebSQL就可以了。在Chrome上，访问*chrome://settings/cookies*，查找*localhost*，然后删除其中的*_ionicstorage*。这样当然会删除在WebSQL中存储的所有数据，不不止是*introShown*标记。

定制主题

当前应用的功能已经100%完成了。我们需要给应用添加一个样式让他比现在看起来更漂亮些。
你应该记得基础部分里面有一些不同的给应用添加样式的方法。我们将给每个组件添加指定的样式，我们将在核心文件中添加一些通用样式，然后在变量文件内覆盖一些SASS变量。如果你跳过了那部分，或者还不大清楚我讲的什么的话，建议你回去重新读一下基础部分的定制主题。
由于我们在Ionic 2中用SASS来制作CSS，我们可以嵌入CSS样式。由于每个页面是他自己的组件，我们可以在组件里有目的的针对元素。尽管如此，我们的每个组件都有独立的*.scss*文件，CSS规则还是全局应用的。如果在其中一个组件中添加如下样式：

```
p {
    font-size: 1.2em !important;
}
```

他将会被应用到应用内的每个组件。
> 修改 **src/pages/intro/intro.scss** 为如下：

```
page-intro {

    ion-slide {
        background-color: #32db64;
    }

    ion-slide img {
        height: 85vh !important;
        width: auto !important;
    }

}
```

我们在*page-intro*里面定义了页面的样式，这样一来他只会在页面添加到DOM的时候应用到元素（因为这个组件有一个*page-intro* 选择器*selector*）里面的元素上，而不是整个应用。这个样式会把滑动组件的背景色设为绿色，将滑动页里面的图片设置为视图高度的85%。
其他组件需要添加其他一些类到模板中，这里我直接po代码了。
> 修改 **src/pages/home/home.html** 为如下：

```
<ion-header>
    <ion-navbar color="secondary">
        <ion-title>
        <img src = "assets/images/logo.png" />
        </ion-title>
        <ion-buttons end>
        <button ion-button icon-only (click)="addChecklist()"><ion-icon name="add-circ
le"></ion-icon></button>
        </ion-buttons>
    </ion-navbar>
</ion-header>

<ion-content>
    <ion-list no-lines>
    <ion-item-sliding *ngFor="let checklist of checklists">
        <button ion-item (click)="viewChecklist(checklist)" class="home-item">
        {{checklist.title}}
        <span class="secondary-detail">{{checklist.items.length}} items</span>
        </button>
        <ion-item-options>
        <button ion-button icon-only color="light" (click)="renameChecklist(checklist)"
><ion-icon
        name="clipboard"></ion-icon></button>
        <button ion-button icon-only color="danger" (click)="removeChecklist(checklist
)"><ion-icon
        name="trash"></ion-icon></button>
        </ion-item-options>
    </ion-item-sliding>
    </ion-list>
</ion-content>
```

> 修改 **src/pages/home/home.scss** 为如下：

```
page-home {

    ion-item-sliding {
        margin: 5px;
    }

    .home-item {
        font-size: 1.2em;
        font-weight: bold;
        color: #282828;
        padding-top: 10px;
        padding-bottom: 10px;
    }

    .secondary-detail {
        display: block;
        color: #cecece;
        font-weight: 400;
        margin-top: 10px;
    }
}
```

我们这里没有做什么疯狂的事情，只是对边距，间隔和颜色做了一些调整。接着我们对checklist页面做相同的事情。

> 修改 **src/pages/checklist/checklist.scss** 为如下：

```
page-checklist {
    ion-item-sliding {
        margin: 5px;
    }

    ion-checkbox {
        font-size: 0.9em;
        font-weight: bold;
        color: #282828;
        padding-top: 0px;
        padding-bottom: 0px;
        padding-left: 4px;
        border: none !important;
    }

    ion-item-content {
        border: none !important;
    }

    ion-checkbox {
        border-bottom: none !important;
    }

    ion-checkbox .item-inner {
        border-bottom: none !important;
    }
}
```

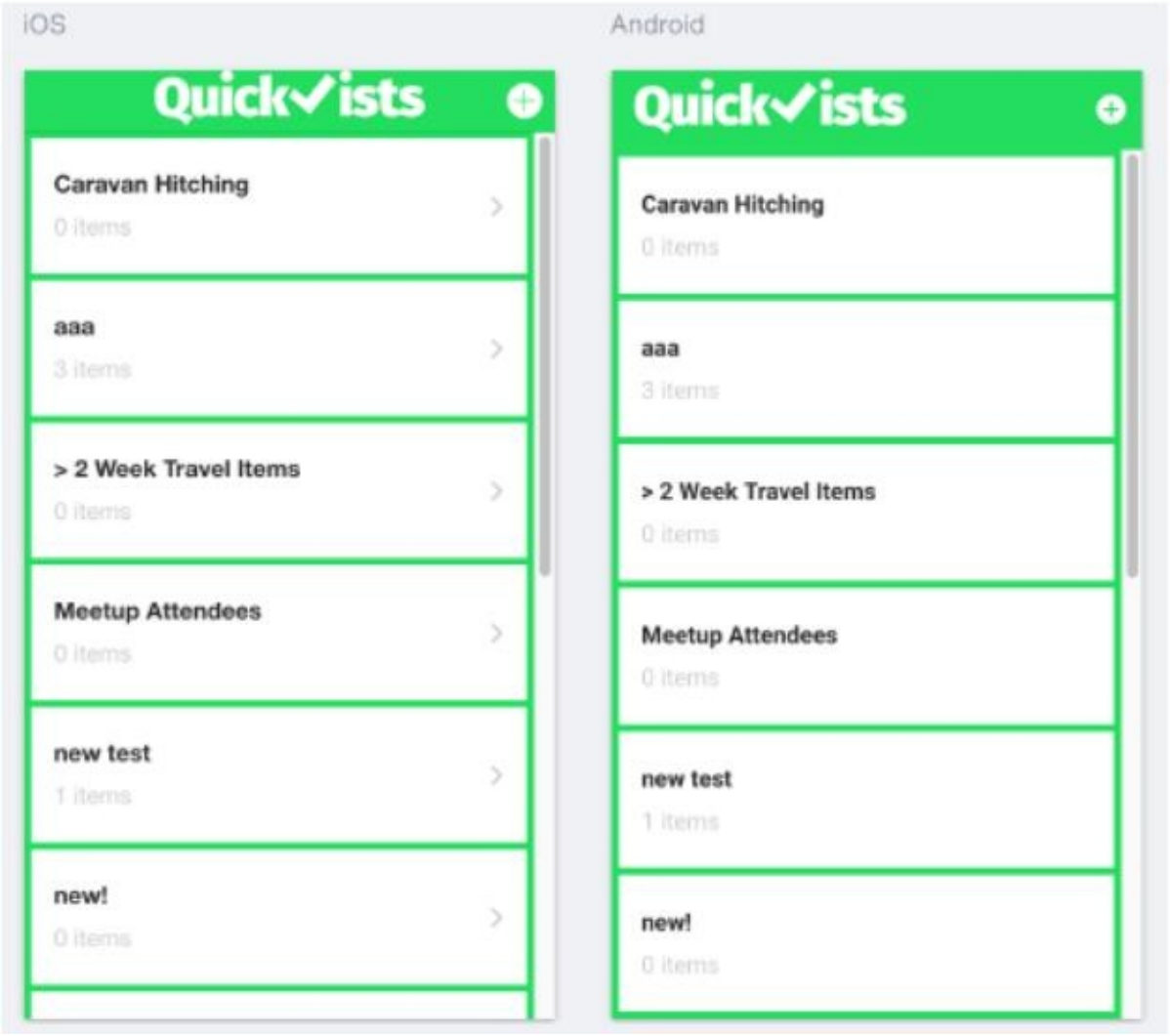
同样，只是一些轻微的调整。我们现在来对整个应用添加一些样式。
> 添加以下样式到 **src/app/app.scss** ：

```
ion-content {
    background-color: #32db64 !important;
}

.logo {
    max-height: 39px;
}

button {
    border: none !important;
}
```

这里我们让整个应用的背景色变成了绿色，我们设置了logo的最大高度，然后移除了按钮的边缘。最后，我们将覆盖一些SASS变量。
> 修改 **src/theme/variables.scss** 命名颜色变量部分为如下：

```
$colors: (
    primary: #387ef5,
    secondary: #32db64,
    danger: #f53d3d,
    light: #f4f4f4,
    dark: #222,
    favorite: #69BB7B
);

$list-background-color: #fff;
$list-ios-activated-background-color: #3aff74;
$list-md-activated-background-color: #3aff74;
$checkbox-ios-background-color-on: #32db64;
$checkbox-ios-icon-border-color-on: #fff;
$checkbox-md-icon-background-color-on: #32db64;
$checkbox-md-icon-background-color-off: #fff;
$checkbox-md-icon-border-color-off: #cecece;
$checkbox-md-icon-border-color-on: #32db64;
```

我们修改了应用的颜色，以及设置了少许iOS和Android特有样式（名字很明显）。Ionic 2最酷的事情之一是他可以很好的操作iOS和Android不同的UI，大部分情况下，他都很完美。当你在iOS和Android上实际看的话，会看到他们之间的不同：



注意：一个比较便利的查看iOS和Android不同的途径是利用Ionic Lab，可以通过命令*ionic serve -l*来激活。可以发现通过Ionic Lab查看应用的时候侧边有滚动条，我觉得这是个bug，他在真实设备上运行的时候不会出现（Chrome Dev Tools模拟器也不会有）。
可以看到，Ionic 2自动适配应用当前运行平台的规范。

总结

就这样！应用现在就宣告完成，看起来也挺不错的。

结论

恭喜你完成了Quick Lists学习指南了。这对入门者小试牛刀来讲是在合适不过的一个选择了，我们通过他学习到了：

- 如何创建，读取，更新和删除数据
- 如何持久化存储数据和重新获取数据
- 如何创建和使用自己的observable
- 如何进行导航和在页面之间传递数据
- 如何创建数据模型

事物都有前进的空间，特别是学习的时候。对照手册固然很好，但是能够自己相处一些东西来就更佳了。希望你现在有足够的背景知识以为应用扩展功能，以下是一些给你参考的扩展想法：

- 用自己的风格重定义应用主题，尝试不同的颜色，间隔和编剧等【简单】
- 给数据模型添加一个'创建日期'用来记录checklist的创建时间，然后将他显示到模板上（别忘了将他存储到存储空间并获取！）【中等】
- 想办法了解一个checklist中有多少项是一完成的，然后展示一个进度条（即 5/7 已完成）。【中等】
- 添加给checklist想做笔记的能力【困难】

记住，当你想要做一些事情的的时候，Ionic 2文档是你最好的朋友。

接下来？

现在，我们完成了一个完成的应用，但是这不是故事的结局。你需要把这个应用弄到眞实设备上运行然后提交到应用商店，这不是个简单的任务。本书最后的部分将带你学习这方面只是，所以可以看去看看。

项目：Giflist

第一课：介绍

Giflist是我制作的第一个Ionic 2应用，他是用非常早的Ionic 2 alpha版本制作的。之后进行了很多的版本更新，本书使用的是Ionic 2 beta版。尽管他是我的第一个原因，但是他始终是我最爱的那个，能够带着你们学习他让我激动不已。
我认为他的是的心头好是因为他是一个很有趣的应用。他是一个制作起来很有趣的应用，因为会涉及到很多有趣的主题，像是使用Reddit API啦，使用HTML5 video啦，他本身也是一个很有趣的应用因为...观赏有趣的GIF动画难道不是很有趣吗？

关于Giflist

Giflist是一个很简单的应用，目的是用户可以输入任何来自reddit的subreddit，应用将从这个subreddit获取和展示GIF。
我假设阅读本书的你知道reddit是什么 -- 但是 -- 我也不能这么去假想，如果你真不熟悉Reddit的话，基本上就是个网站，用户可以提交一些有趣的链接供其他用户投票。一个“subreddit”基本上就是reddit上的一个字目录，以下是一些比较受欢迎的：

- gifs
- askreddit
- worldnews

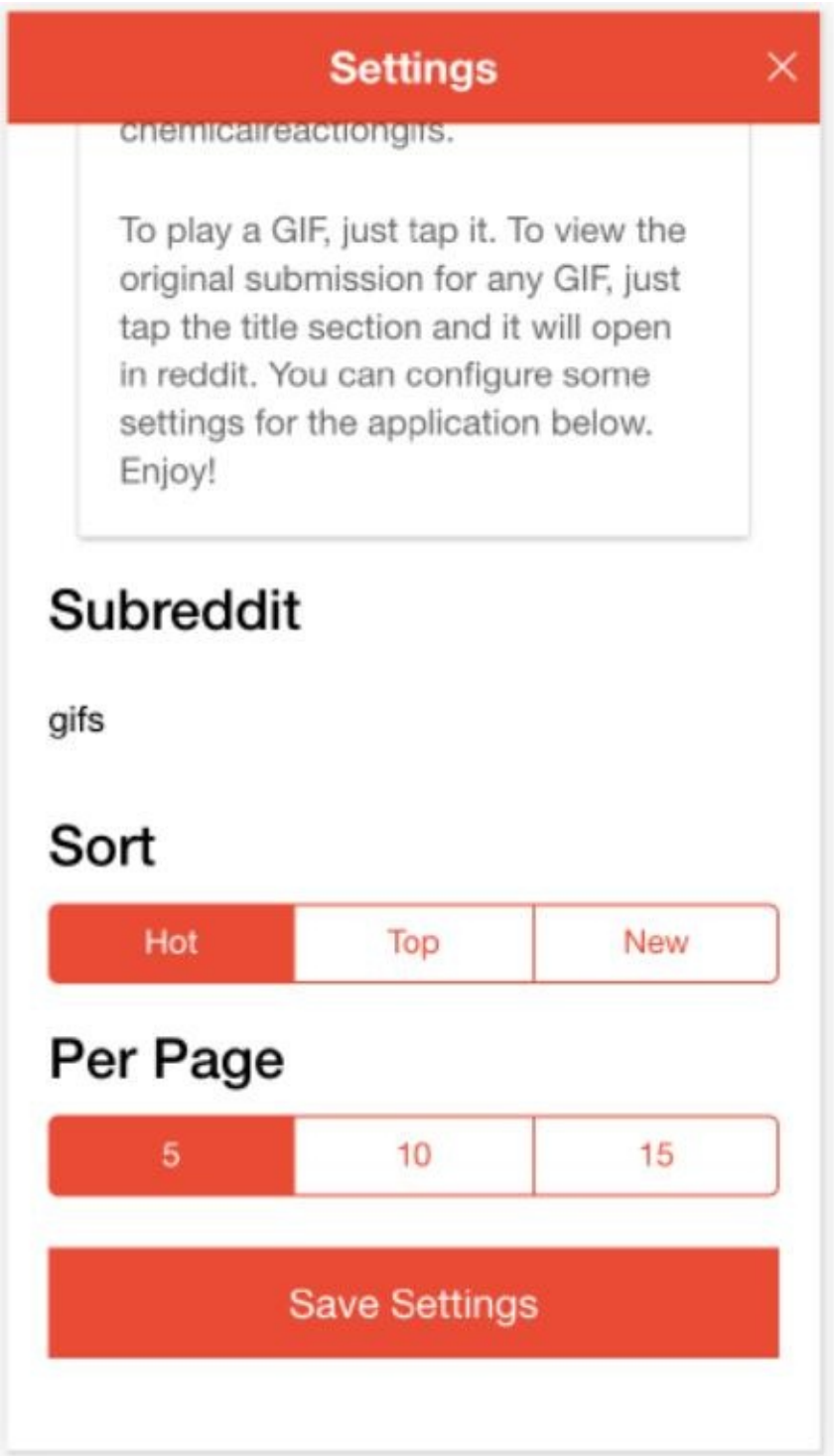
在从reddit拉取GIF的同时，用户也可以提供一些设置来配置应用的首选项。虽然理念很简单，在学习过程中还是可以学到一些有趣的东西，包括：

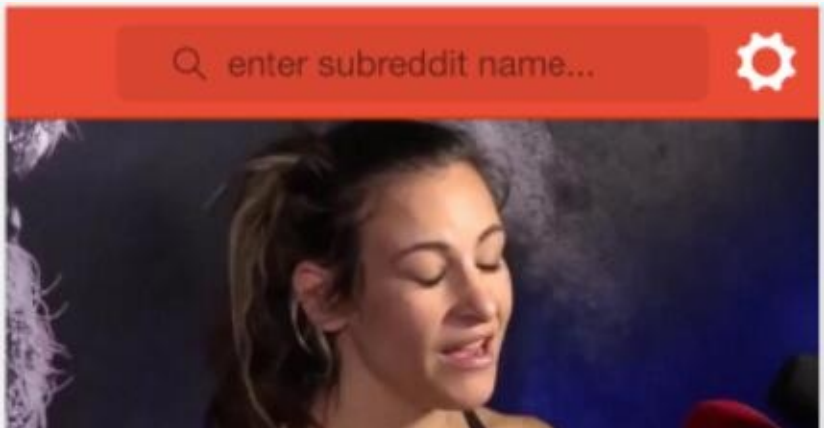
- 从第三方API获取数据
- 数据存储
- 定制主题
- 列表
- 模态框
- 数据模型
- HTML5 VIDEO

以下是原因的一些特性：

- 用户可以输入任何的subreddit
- 会展示一个无尽的GIF列表（结社GIF有尽）
- 用户可以通过点击GIF来播放他
- 用户可以设置一下选项：默认的subreddit，排序方式，每页展示的GIF数
- 下次再使用应用的时候可以记起之前的设置

以下是一些应用成品的截图：





课程结构

1. 准备工作
2. 列表页
3. Reddit API和HTML5 VIDEO
4. 设置
5. 定制主题

准备好了吗？

了解了我们的目的之后，我们现在可以开始来制作他了！

第二课：准备工作

本课是在旅程继续进行之前的一些准备工作。我们要生成应用，设置所有组件和需要用到的Cordova插件。完成本课之后我们应该有一个万事俱备的项目骨架，可以接着进行编码工作。开始新项目的第一准则是确保使用的是最新版的Ionic和Cordova，如果最近没有更新过的话可以运行如下命令：

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

或者

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

如果在安装Ionic或者生产新项目的时候遇到任何问题的话，检查下是否安装了最新版的Node。安装完之后，再次安装ionic之前请运行：

```
npm uninstall -g ionic npm cache clean
```

生成新应用

本应用将使用空白初始模板，跟名字说的一样，就是个空的Ionic项目。但是会有一个内置的页面名为**home**，我们下节课中将用作列表显示页。
> 运行如下命令生成新项目：

```
ionic start giflist blank --v2
```

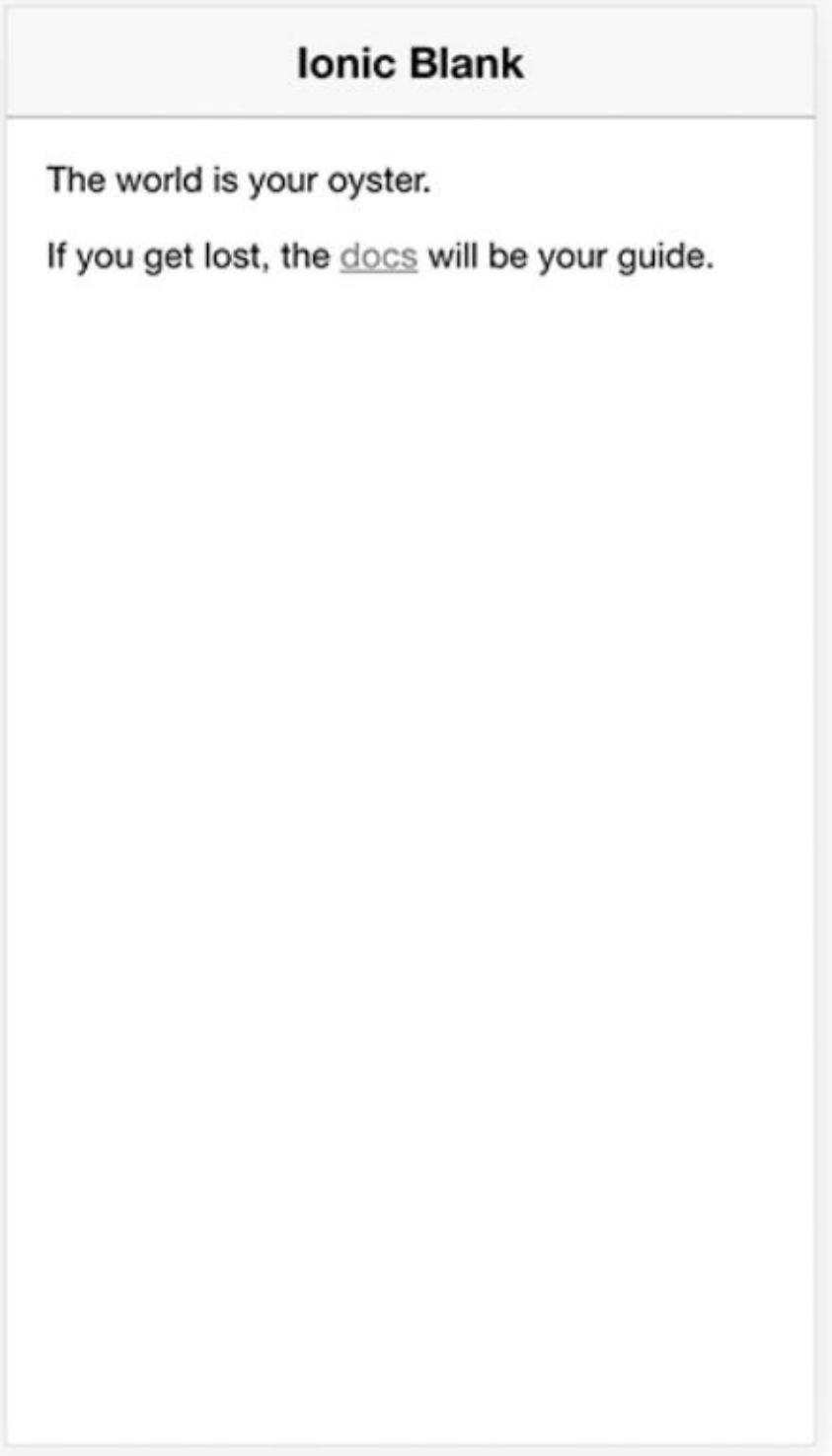
> 运行如下命令将新生成的项目作为当前目录：

```
cd giflist
```

现在可以在你中意的编辑器中打开这个项目了。通过以下命令可以预览创建的应用：

```
ionic serve
```

看起来是这样的：



创建需要的组件

应用的架构很简单，我们会有两个页面：列表页和设置页。我们已经有**home**组件作为我们的列表页了，所以我们只需要创建**Setting**页就行了。

> 运行如下命令生成 **Settings** 页面：

```
ionic g page Settings
```

创建需求的服务

跟之前的两个页面一样，我们需要创建数据服务来操作用户设置的保存和获取：
> 运行如下命令生成 **Data** 提供者：

```
ionic g provider Data
```

这个提供者将负责为我们从reddit拉取数据：
> 运行如下命令生成 **Reddit** 提供者：

```
ionic g provider Reddit
```

在**App Module**里面添加 页面 **&** 服务

为了能够在项目里面可以使用这些页面和服务，我们需要将它们添加到**app.module.ts**文件里。所有我们自己创建的页面都需要添加到*declarations*数组和*entryComponents*数组里，所有我们创建的数据提供者都需要添加到*providers*数组，其他自定义组件或者管道（pipe）只需要添加到*declarations*数组即可。我们的数据模型只是一个简单的类，我们需要在任何地方使用，所以不用在模组里面设置。
> 修改**src/app/app.module.ts**到以下：

```
import { NgModule } from '@angular/core';
import { IonicApp, IonicModule } from 'ionic-angular';
import { MyApp } from './app.component';
import { Storage } from '@ionic/storage';
import { HomePage } from '../pages/home/home';
import { SettingsPage } from '../pages/settings/settings';
import { Reddit } from '../providers/reddit';
import { Data } from '../providers/data';

@NgModule({
    declarations: [
        MyApp,
        HomePage,
        SettingsPage
    ],

    imports: [
        IonicModule.forRoot(MyApp)
    ],

    bootstrap: [IonicApp],

    entryComponents: [
        MyApp,
        HomePage,
        SettingsPage
    ],

    providers: [Storage, Data, Reddit]
})
export class AppModule {}
```

注意，我们除了自己创建的*Data*提供者之外，我们还加入了一个*Storage*。Storage是Ionic提供的，可以通过它保存和获取数据 -- 我们后续会用到。

添加需要的平台

在给指定平台制作应用之前，你需要将它们添加到你的项目。
> 运行以下命令添加**iOS**平台：

```
ionic platform add ios
```

> 运行以下命令添加**Android**平台：

```
ionic platform add android
```

添加需要的Cordova插件

这个应用将会用到不同的Cordova插件。记住，Cordova插件只能在真实设备上运行。我将在添加他们的时候解释。
> 运行以下命令添加**SQLite**插件：

```
ionic plugin add cordova-sqlite-storage
```

这个插件让你可以访问本地存储SQLite数据库。我们在此应用中添加他的原因是Ionic本地存储服务可以使用插件提供的稳定输出存储。

> 运行以下命令添加**App Browser**插件：

```
ionic plugin add cordova-plugin-inappbrowser
```

这个插件让我们可以通过他提供的webview来气从外部网站。我们将通过这个插件在应用中让用户在浏览器中查看原先的Reddit GIF。
> 运行以下命令添加**Status Bar**插件：

```
ionic plugin add cordova-plugin-statusbar
```

我们给所有项目添加此插件用来在应用中控制状态栏（设备屏幕顶部的状态条，包括时间，电池信息等等）。
> 运行以下命令添加**Splash Screen**插件：

```
ionic plugin add cordova-plugin-splashscreen
```

此插件允许我们控制闪屏（打开应用的时候的全屏画面）。
> 运行以下命令添加**Keyboard**插件：

```
ionic plugin add ionic-plugin-keyboard
```

这个插件允许我们控制软键盘。
> 运行以下命令添加**Whitelist**插件：

```
ionic plugin add cordova-plugin-whitelist
```

所有应用会用到这个插件，他定义了应用里可以加载什么样的资源。没有他的话，你尝试加载的资源都会不成功。
添加了这个插件后，你也需要到**index.html**中定一个一个“Content Security Policy”。我们将添加一个非常宽松的策略实际上允许我们加载任何资源。基于你的应用，你可以提供一个更严格的策略，但是对于开发而言开放性策略就可以了。
> 修改 **src/index.html**文件，添加一下**meta**标签：

```
<meta http-equiv="Content-Security-Policy" content="font-src 'self' data:;
img-src * data:; default-src gap://ready file://* *; script-src 'self'
'unsafe-inline' 'unsafe-eval' * ; style-src 'self' 'unsafe-inline' *">
```

> 运行以下命令添加**Crosswalk**插件：

```
ionic plugin add cordova-plugin-crosswalk-webview
```

这个另一个每个应用都要添加的插件，但是你也可以先不添加。添加了这个插件后，在你编译Android的时候将会使用“Crosswalk”。Android有很多问题，特别是老设备，因为有太多不同的团建版本，不同的版本有不同的浏览器（记住，鉴于我们是制作HTML5应用，他实际上就是一个搭载的浏览器用来运行我们的应用）。Crosswalk做的是将一个现代的浏览器打包到应用中，这样一来应用无论是运行在什么设备上，都会使用相同的浏览器来运行，并且Crosswalk浏览器可以很好改善执行效率。
唯一的不足之处就是你的应用尺寸明显的变大了很多。总体上，我觉得这很值得，我也建议你使用他，如果你接受不了的话，也可以不用。更多关于Crosswalk Project的信息，请参考网站：https://crosswalk-project.org/

设置图片

制作此应用的时候，会用到一些图片。你下载的包里面已经包含了这些图片，但是你需要去生成的项目里面设置好他们。
> 将下载包 **src/assets**文件夹下面的**images**文件夹复制到应用里的 **src/assets**下面

总结

就这样！我们设置好了，准备好继续前几，现在我们开始进入到有趣的部分了。

（译者：以上内容从上一章拷过来的，经过对比发现并无差别，所有后续极有可能也会拷）

第三课：列表页

上一节课我们做好了所有的准备工作，我们现在就要开始进行实际创作了。本课我们将聚焦于**Home**页来把它改造成用来展示GIF回馈的列表页。

Reddit提供者

我们将要创建的布局严重的依赖从reddit拉取的数据，我们决定了创建一个提供者来为我们做这些操作。我们现在不会进入reddit提供者的实现细节，但是我们现在得设置好我们最终将要用到的函数，这样可以在本课中用上他们。
> 修改 **src/providers/reddit.ts**为如下：

```
import { Injectable } from '@angular/core';
import { Http } from '@angular/http';
import 'rxjs/add/operator/map';

@Injectable()
export class Reddit {
    settings: any;
    loading: boolean = false;
    posts: any = [];
    subreddit: string = 'gifs';
    page: number = 1;
    perPage: number = 15;
    after: string;
    stopIndex: number;
    sort: string = 'hot'
    moreCount: number = 0;

    constructor(public http: Http) {

    }

    fetchData(): void {

    }
}
```

这几本书就是这个提供者的股价，他定义了大量的成员变量和*fetchData*函数用于从reddit获取数据。成员函数的作用是什么现在可能不是很明显，我们还是（按顺序）了解一下吧：

- 用户提供的设置settings
- 是否加载了当前新的GIF

- 当前加载完成的所有的GIF的入口
- 当前的subreddit
- 当前页（即，用户点击“Load More”的次数）
- 每页展示的GIF的数量
- 最后一个从Reddit拉取的帖子的引用（这样我们就知道下次从哪个页面开始）
- 用于存储帖子数组的长度
- GIF排列依据
- 应用会持续向reddit拉取数据直到够一页的数据，moreCount告诉它加载更多是具体多少（即，如果在请求API20次还不够的话）

我们希望尽快完成这个提供者，但是现在我们还是先完成home页面布局。

布局

在开始制作这个布局之前，也就是**home.html**，我们先看一下效果图：

这是一个很简单的布局，顶部一个导航条，包含一个搜索条一个设置按钮（用于启动Settings页面）。在他下面是一系列的从Reddit返回的GIF列表。还有截屏里面没有显示的列表底部的‘Load More’按钮，用户点击这个按钮的时候将会加载下一页的GIF。
首先我们看一下整个模板，然后分成各个小块来详细讲解。
> 修改 **src/pages/home/home.html**为如下：

```
<ion-header>
    <ion-navbar color="secondary">
        <ion-title>
            <ion-searchbar color="primary" placeholder="enter subreddit name..."></ion
-searchbar>
        </ion-title>
        <ion-buttons end>
            <button ion-button icon-only (click)="openSettings()"><ion-icon name="sett
ings"></ion-icon></button>
        </ion-buttons>
    </ion-navbar>
</ion-header>
<ion-content>
    <ion-list>
        <ion-item no-lines>
        GIF GOES HERE
        </ion-item>
        <ion-list-header>
        TITLE GOES HERE
        </ion-list-header>
        <ion-item *ngIf="redditService.loading" no-lines style="text-align:center;">
            <img src="assets/images/loader.gif" style="width: 50px" />
        </ion-item>
    </ion-list>
    <button ion-button full color="light" (click)="loadMore()">Load More...</button>
</ion-content>
```

第一部分设置了导航条：

```
<ion-navbar color="secondary">
    <ion-title>
        <ion-searchbar color="primary" placeholder="enter subreddit name..."></ion-sea
rchbar>
    </ion-title>

    <ion-buttons end>
        <button ion-button icon-only (click)="openSettings()"><ion-icon name="settings"
></ion-icon></button>
    </ion-buttons>
</ion-navbar>
```

我们给添加了*secondary*是属性这样后续会将他样式调整到使用secondary颜色。
我们用了，这个组件通常是用来在navbar中展示标题的，这里我们用来将搜索条调整到navbar的居中位置。通常每个搜索条会有一个单独的工具栏这样它会占用整个空间，但是我不想让屏幕看起来很乱所以这里用它来稍微调整一下。为了能够让他恰当的去工作，我们后续会添加一些自定义样式。我们也给搜索条提供了primary属性这样他会用primary颜色。
然后，我们使用来防止我们的设置按钮，这个按钮会启动设置页面。使用end指令可以将按钮

放到右边，如果想把按钮放到左边的话那么就用start指令。我们也给按钮添加了一个*(click)*监听器这样在点击按钮的时候会调用*openSettings*函数。我们现在没有创建这个函数所以现在点击的时候什么都不会发生，但是我们稍后会在**home.ts**中定义。
接下来我们定义了主题内容区域：

```
<ion-list>
    <ion-item no-lines>
    GIF GOES HERE
    </ion-item>

    <ion-list-header>
    TITLE GOES HERE
    </ion-list-header>
    <ion-item *ngIf="loading" no-lines style="text-align: center;">
        <img src="assets/images/loader.gif" style="width: 50px" />
    </ion-item>
</ion-list>
```

列表是移动应用中使用最多的组件之一。Ionic中你可以通过创建一个然后给其中添加任意数量的来创建一个列表。目前我们只有一个项，后续将改为自动循环每个需要显示的GIF。同时我们也用到了来创建一个头来展示GIF的标题，同时也给项添加了*no-lines*属性这样列表项不会有边缘显示了。
跟GIF项一样，我们将在列表的底部添加一个额外的项，其中包含了一个加载动画。他将用于在获取新的GIF的过程中显示一个旋转动画，但是由于他只会在发生加载的时候显示，我们使用了 **ngIf* 指令来控制他的显示时机。本案例中，加载动画只会在*loading*为true的时候才显示（这个值会后面在类中定义，然后在加载前和加载完成后对他进行更改）
模板中剩下的代码是一个加载更多按钮：

```
<button ion-button full color="light" (click)="loadMore()">Load More...</button>
```

这里没什么惊心动魄的东西，给这个组件提供了一个*light*指令以改变他的色值，有一个*(click)*函数设置点击的时候调用类定义中的*loadMore()*函数。

类定义

在模板定义里我们解决了我们的“视图”，现在现在需要去制作类定义来处理列表页的逻辑。这里用于定义模板里面引用到的所有函数，以及其他一些页面运行的代码。
同理，我们先贴出所有代码然后一点一点的来解释。
> 修改 **src/pages/home/home.ts** 为如下：

```
import { Component } from '@angular/core';
import { ModalController, Platform } from 'ionic-angular';
```

```
import { Keyboard } from 'ionic-native';
import { SettingsPage } from '../settings/settings';
import { Data } from '../../providers/data';
import { Reddit } from '../../providers/reddit';
import { FormControl } from '@angular/forms';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/debounceTime';
import 'rxjs/add/operator/distinctUntilChanged';

@Component({
    selector: 'page-home',
    templateUrl: 'home.html'
})
export class HomePage {
    subredditValue: string;
    constructor(public dataService: Data, public redditService: Reddit, public  modalC
trl: ModalController, public platform: Platform) {

    }

    ionViewDidLoad(){
        this.platform.ready().then(() => {
            this.loadSettings();
        });
    }

    loadSettings(): void {
        console.log("TODO: Implement loadSettings()");
    }

    showComments(post): void {
        console.log("TODO: Implement showComments()");
    }

    openSettings(): void {
        console.log("TODO: Implement openSettings()");
    }

    playVideo(e, post): void {
        console.log("TODO: Implement playVideo()");
    }

    changeSubreddit(): void {
        console.log("TODO: Implement changeSubreddit()");
    }

    loadMore(): void {
        console.log("TODO: Implement loadMore()");
    }
}
```

显然新加的代码不少，即使只是基本类定义看起来也挺复杂的。我们先走一遍。
首先是**import**语句：

```
import { Component } from '@angular/core';
import { ModalController, Platform } from 'ionic-angular';
import { Keyboard } from 'ionic-native';
import { SettingsPage } from '../settings/settings';
import { Data } from '../../providers/data';
import { Reddit } from '../../providers/reddit';
import { FormControl } from '@angular/forms';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/debounceTime';
import 'rxjs/add/operator/distinctUntilChanged';
```

这里有很多导入，因为我们把后续会用到的所有东西都导入进来了，但是对于后面大部分案例来讲，这算是少的了。
前面的非常基础，**ModalController**允许我们创建一个模态页展示到当前页面的顶部（译者：不是页面的顶部，是覆盖整个页面的上面），**Platform**允许我们设备加载完成之后执行一些操作。之后，我们导入了Keyboard插件（稍后使用）。
我们也导入了之前创建的**SettingsPage**（目前还没完成），导入了**Data**提供者（当前也没完善），还有未完成的**Reddit**提供者。**FormControl**允许我们为输入框创建一个“FormControl”（让我们可以访问Observable）。**rxjs**导入的是RxJS库 -- 令人烦恼的是，你需要自己导入Observable的操作符，所以接下来导入了。操作符的使用在使用的时候再讨论。
接下来的是构造器*constructor*函数和*ionViewDidLoad*函数。构造器函数的类的重要组成部分，因为他是类实例化的时候第一个执行的函数。在其中我们可以注入和设置需要用到的组件或者服务，也是想要立即执行一些函数的最佳点。始终记住，最佳体验是不要在构造器里面做太多的“工作”，这样一些代码你可以放到*ionViewDidLoad()*周期函数里面去（这个函数在页面加载完成之后第一个执行）。

```
subredditValue: string;

constructor(public dataService: Data, public redditService: Reddit, public modalCtrl:
ModalController, public platform: Platform) {

}
ionViewDidLoad(){

    this.platform.ready().then(() => {
        this.loadSettings();
    });
}
```

在*ionViewDidLoad*中我们设置了注入服务的引用（我们的Data和Reddit），在构造器的顶部我们设置了一个成员变量用于绑定到模块中的搜索条的输入框。我们也调用了*loadSettings()*函数将会从存储中加载用户设置。重点是我们是在平台准备好之后再调用的。
我们来看看剩下的代码：

```
loadSettings(): void {
        console.log("TODO: Implement loadSettings()");
    }

    showComments(post): void {
        console.log("TODO: Implement showComments()");
    }

    openSettings(): void {
        console.log("TODO: Implement openSettings()");
    }

    playVideo(e, post): void {
        console.log("TODO: Implement playVideo()");
    }

    changeSubreddit(): void {
        console.log("TODO: Implement changeSubreddit()");
    }

    loadMore(): void {
        console.log("TODO: Implement loadMore()");
    }
```

剩下的只是一大堆的函数。这些方法要么是模板调用的，要么是里面其他地方调用的（构造器或者其他方法）。很明显他们现在都是白板，后面会对他们进行扩展。

使用Observable来控制搜索

我们接下来用Observable。我们已经在基础部分详细探讨过什么是Observable，如果你忘得差不多了的话，建议现在回去读一遍获取数据，Observable和Promise部分。
大部分Observable的使用就是简单的订阅Observable的返回值，例如使用Http服务获取数据（这个应用的下一部分就会用到了）。所以基本你不用自己创建一个Observable。但是如果你想要的话，还是可以用得上他提供的一些其他的方法。
我们将用到之前导入的**FormControl**服务来创建一个“FormControl”这样就会给咱们提供一个Observable。FormControl跟[(ngModel)]提供的创想数据绑定的工作方式很像，他们都是将类定义中的变量和模板中的输入域捆绑到一起。接下来我们做一些变更。
> 修改 **src/pages/home/home.html** 为如下：

```
<ion-searchbar color="primary" placeholder="enter subreddit name..." [(ngModel)]="subr
edditValue" [formControl]="subredditControl" value=""></ion-searchbar>
```

我们这里所作只是添加了一个[formControl]，他将提供给稍后创建的**Control**使用。接下来，我们来对类的构造器做一些改动。

> 修改 **src/pages/home/home.ts** 的**constructor**和**ionViewDidLoad**为如下：

```
subredditValue: string;
subredditControl: FormControl;

constructor(public dataService: Data, public redditService: Reddit, public modalCtrl:
ModalController, public platform: Platform) {
    this.subredditControl = new FormControl();
}

ionViewDidLoad(){

    this.subredditControl.valueChanges.debounceTime(1500)
    .distinctUntilChanged().subscribe(subreddit => {
        if(subreddit != '' && subreddit){
            this.redditService.subreddit = subreddit;
            this.changeSubreddit();
            Keyboard.close();
        }
    });

    this.platform.ready().then(() => {
        this.loadSettings();
    });
}
```

首先我们创建了一个新的**FormControl**，然后订阅了他提供的*valueChanges*Observable。如果你看过基础部分的Observable的话，那么这里看起来没什么奇怪的。我们订阅**subscribe**了observable这样每次他提交新的值的时候都会运行这段代码。这里比较奇怪的地方是*valueChanges.debounceTime(1500).distinctUntilChanged()*。基本上，我们可以任意链接需要数量的操作符（因为每个函数都是返回的Observable，所以还是可以订阅）他们每一个的做的事情都不一样。例如，当我们这么做的话：

```
this.subredditControl.valueChanges.debounceTime(1500).subscribe
```

我们只有在输入发生变更且1.5秒内没有新的输入的话才运行代码。这防止我们频繁提交无意义的请求到API。例如，当用户输入‘chemicalreactiongifs’的时候，代码将会触发‘c’，然后‘ch’，然后‘che’，然后‘chem’等等直到完成的单词输入完成。他不止是频繁向发送无用的查

询，带来的是结果列表频繁刷新导致极差的用户体验。通过添加弹性时间，代码只会在完整的字符串输入完成之后运行一次（假设用户字符之间输入间隔用不了1.5秒）。
最后，我们添加了最终的操作符：

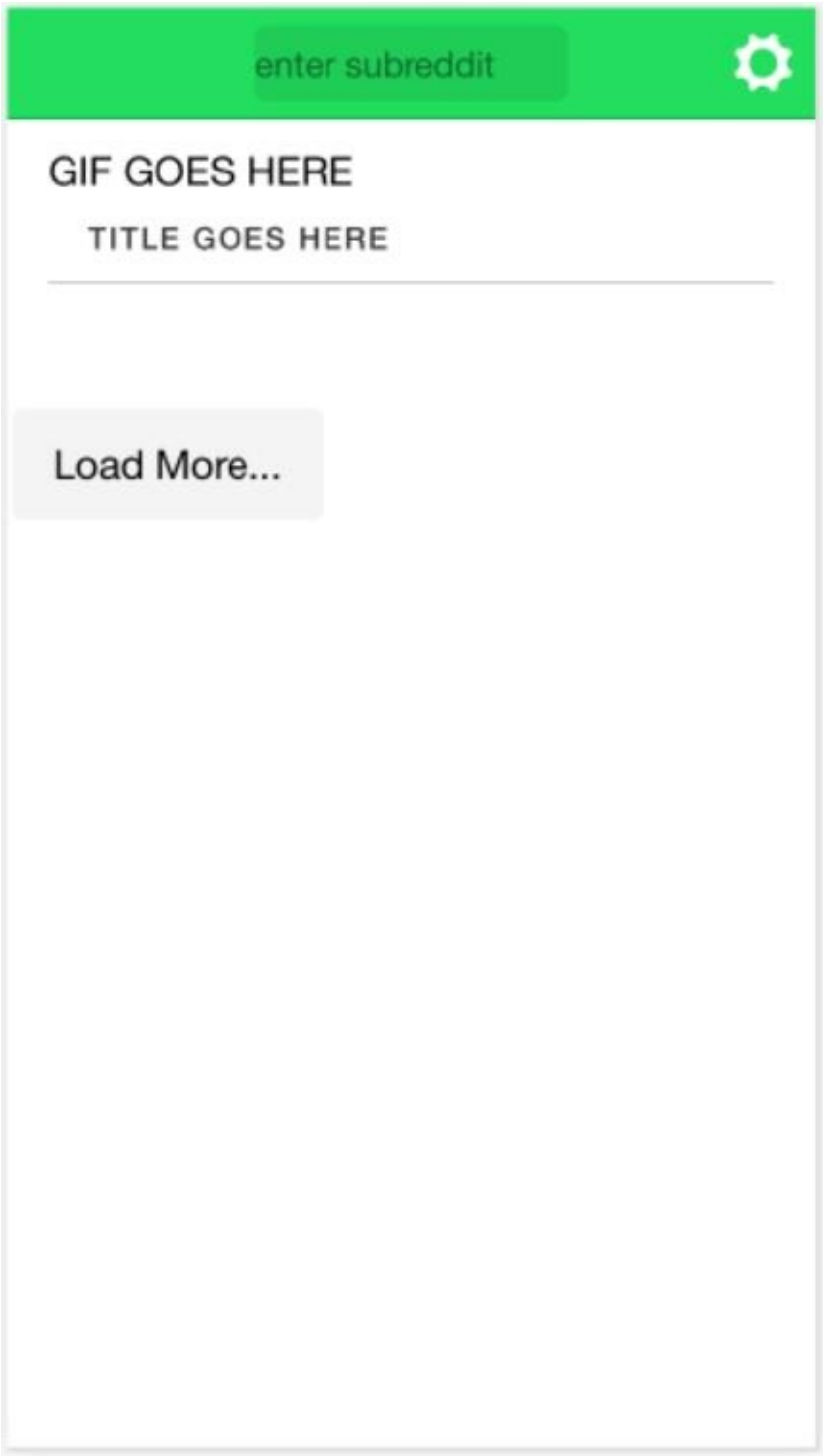
```
this.subredditControl.valueChanges.debounceTime(1500).distinctUntilChanged().subscribe
```

这样只有值和上次的值不一样的时候才会运行代码。所以当用户输入‘gifs’，点击回退键将改成‘gif’，然后又假设‘s’将他变成‘gifs’，什么事情都不会发生。我们不需要重新为‘gifs’加载数据，因为他已经是了。
最终结果是当值发生改变的时候代码才会触发，用户现在还没有输入，输入值和之前的一样。触发的代码只是简单的检查是否提供了一个空值，然后通过更改**Reddit**的*subreddit*成员变量，调用他的*changeSubreddit()*函数来改变活跃的subreddit。我们也调用了Keyboaid插件的*close()*方法；由于我们做的事情违反了输入域的规则，键盘不知道何时关闭，所以我们手动来关闭。
这可能是应用最迷惑的部分，特别是当你刚知道Observable的情况下。你也知道他允许我们很轻松的创建一些很有用的功能，但是我们也可以很简单的使用普通 ngModel 方式和使用一个按钮来触发搜索行为来替换他。

总结

本课中我们完成了一个非常好的开头，我们在制作一些比较酷的这条路上也渐渐上到。我们也有了一个很好的基础架构这样允许我们在下一课中在它只是轻松的制作功能。如果你现在通过以下命令运行应用的时候：

```
ionic serve
```



应该可以看到这样的效果图：

好丑哇。甚至你还可以在控制台中看到一些错误。相信我，最后一切都会解决的！下一棵将会从reddit拉取一些真实数据。

第四课：Reddit API和HTML5 Video

Giflist最有趣的地方是他里面没有一个GIF文件。GIF文件尺寸很大，加载很慢，虽然用户更在意他们的数据，但是这个也是个很大的问题。
所以我们要做的是拉取GIF提供的**.webm**或者**.gifv**格式。这意味着我们不会展示GIF，我们将展示的是**video**视频。
我们将使用HTML5的*video*标签来展示这些视频。始终记住，用HTML5来制作移动应用就可以使用HTML5的所有功能。非常典型的一个例子是**Geolocation** -- 我们可以使用本机【native】API来访问设备的GPS，但是在网页上我们也可以使用HTML5自带的Geolocation API。基本上，任何网页上可以做到的事情，移动应用上也可以做到（很明显，我们可以做到更多，因为我们可以访问本机功能）。
在实现此功能之前，我们先来熟悉一下HTML5 Video。

HTML5 Video在iOS和Android上的行为

使用类似Ionic这样的框架意味着他们帮我们处理了平台之间的差异性，但是当制作跨平台应用的时候，还是会遇到一些平台差异性相关的问题。
首先，对于*video*元素有一些需要知道的事情：

- 他可以全屏展示也可以嵌入使用
- 他有一个**poster**属性用于在视频加载之前作为封面展示
- 可以控制的视频的：控件是否展示，视频是否自动播放，是否嵌入页面播放

重点记住，*video*元素根据运行平台的不同，他的行为会有所不同。

- iOS上视频默认是全屏播放，但是也可以通过**webkit-playinline**属性来强制默认嵌入页面播放。同时，这也不是所有iOS设备通用的。在小的设备上，即使你指定了**webkit-playinline**属性，他还是会默认全屏播放（基本上，这个是没法解决的）。
- Android设备上默认是嵌入页面播放，但是可以设置默认全屏播放。

知道了这些不同，我们就需要找出如果解决这些问题。我们基本上有两个可选项：

1. 接受默认行为，不同平台使用相同的代码
2. 检查运行平台，运行不同代码来达到需求

个人而言，我更希望嵌入网页播放视频。但是由于在iOS小型设备上会默认全屏，我觉得还是用默认的行为好些。这意味着在iOS和Android上表现会有所不同，但是我觉得两者都很完美都可以接受，同时可以保持我们的代码简单整洁。
好了，我们开始工作了。我们将实现**home.ts**里面的一些函数定义然后一个个的讲解。

从Reddit获取数据

我们从最复杂最有趣的函数开始，同时也是整个应用最重要的核心功能：*fetchData()*。我们先添加代码然后讲解。

> 修改 **src/app/providers/reddit.ts** 的 **fetchData** 函数为如下：

```
fetchData(): void {
    //基于用户当前偏好组装URL来访问API
    let url = 'https://www.reddit.com/r/' + this.subreddit + '/' + this.sort + '/.json
?limit='+ this.perPage;
    //如果我们不是在第一页的话，我们需要加上after参数才能得到新的结果
    //这个参数基本上就是讲"把 AFTER 这个帖子的帖子给我"
    if(this.after){
        url += '&after=' + this.after;
    }
    //我们现在拉取数据，所有要将loading变量设为 true
    this.loading = true;
    //向指定的URL发起请求然后订阅他的 response
    this.http.get(url).map(res => res.json()).subscribe(data => {
        let stopIndex = this.posts.length;
        this.posts = this.posts.concat(data.data.children);
        //循环所有的 NEW 帖子。
        //我们倒序循环的原因是因为需要移除一些项。
        for(let i = this.posts.length - 1; i >= stopIndex; i--){
            let post = this.posts[i];
            //添加一个新属性用于切换单个帖子的加载动画
            post.showLoader = false;
            post.alreadyLoaded = false;
            //给 NSFW 帖子添加 NSFW 印记
            if(post.data.thumbnail == 'nsfw'){
                this.posts[i].data.thumbnail = 'images/nsfw.png';
            }
            /*
            * 移除所有非 .gifv 或者 .webm 格式的帖子，然后将保留下来的帖子转换成.mp4文件
            *
            */
            if(post.data.url.indexOf('.gifv') > -1 ||  post.data.url.indexOf('.webm')
> -1){
                this.posts[i].data.url = post.data.url.replace('.gifv', '.mp4');
                this.posts[i].data.url = post.data.url.replace('.webm', '.mp4');
                //如果有缩略图的话，将他指定到 post 的 'snapshot'
                if(typeof(post.data.preview) != "undefined"){
                    this.posts[i].data.snapshot =  post.data.preview.images[0].source.
url.replace(/&amp;/g, '&');
                    //如果 snapshot 未定义的话, 将他指定为空这样就不会显示一个破裂图
                    if(this.posts[i].data.snapshot == "undefined"){
                        this.posts[i].data.snapshot = "";
                    }
                }
                else {
                    this.posts[i].data.snapshot = "";
```

```
                }
            }
            else {
                this.posts.splice(i, 1);
            }
        }
        //如果没有得到够一页的数据那么继续获取GIF
        //但是，如果连续20次都没获取足够的数据的话就放弃
        if(data.data.children.length === 0 || this.moreCount > 20){
            this.moreCount = 0;
            this.loading = false;
        } else {
            this.after = data.data.children[data.data.children.length - 1].data.name;
            if(this.posts.length < this.perPage * this.page){
                this.fetchData();
                this.moreCount++;
            }
            else {
                this.loading = false;
                this.moreCount = 0;
            }
        }
    }, (err) => {
        //静默失败，此时加载旋转动画会持续显示
        console.log("subreddit doesn't exist!");
    });
}
```

这个函数还是蛮大的。我在里面加入了一些注释来帮助理解，但是我们还是来详细讲解一下每片代码。

首先，我们创建了用来向Reddit API获取数据的URL。我们用到了用户当前设置的**subreddit**，**sort**和**perPage**。你可以任意修改这些值，他将会输出成对应的JSON。如果有提供**after**的话，那么他也会输出到JSON。这就是Reddit API的“分页”方式，如果用户点击“Load More”按钮三次的话，我们只会返回第三页的帖子，即，如果每页5个的话就是10-15. 你可以给Reddit API提供一个帖子“name”，他只会返回这个帖子后面的帖子。

然后我们用这个URL来发起Http请求，然后在将Reddit返回结果JSON字符串JSON话成一个对象之后，订阅他的Observable。

之后，我们可以循环返回的数据对其施展魔法。我们不需要循环存储在*this.posts*变量中的每个帖，因为其中大部分都“处理过”，我们只要处理新加载的就可以了 -- 所以我们创建了一个“stopIndex”作为*this.posts*数组的长度，然后我们将新帖子加入其中。

循环处理新加载的帖子的时候，我们做了如下处理：

- 将*.gifv*和*.webm*转换成*.mp4*
- 给帖子新增一个‘showLoader’变量用来切换加载动画的显示
- 给NSFW帖子指定NSFW标志
- 如果帖子有预览图的话给他添加一个预览图属性

循环完之后，我们就可以得到一个格式合格的帖子数组来，可以用在列表显示中了。还有一个重要的待作步骤。如果我们每页从Reddit API加载10个帖子的话，但是其中只有3个帖子适应GIF，我们的页面尺寸将变成3。这对于用户来讲就不是很友好了。

解决这个问题的方法是我们将在*fetchData()*内递归多次调用*fetchData()*函数。这样我们的帖子数组里面将会有越来越多的帖子直到填满10个（或者当前页面尺寸）为止。同时我们也的设置一个限制以防无限调用这个函数，所以在调用了20次之后还没有填满的话我们会自动放弃。

同时我们也给*http.get*添加了错误处理器。如果请求成功将会运行上面讨论的代码，如果失败的话（即用户想要访问的subreddit返回结果404），那么将会进入错误处理器而不是上面的代码。

现在我们有了GIF加载到应用中，我们就可以将真实数据展示到列表中。但是，首先，我们的更新模板来用于展示真实数据。

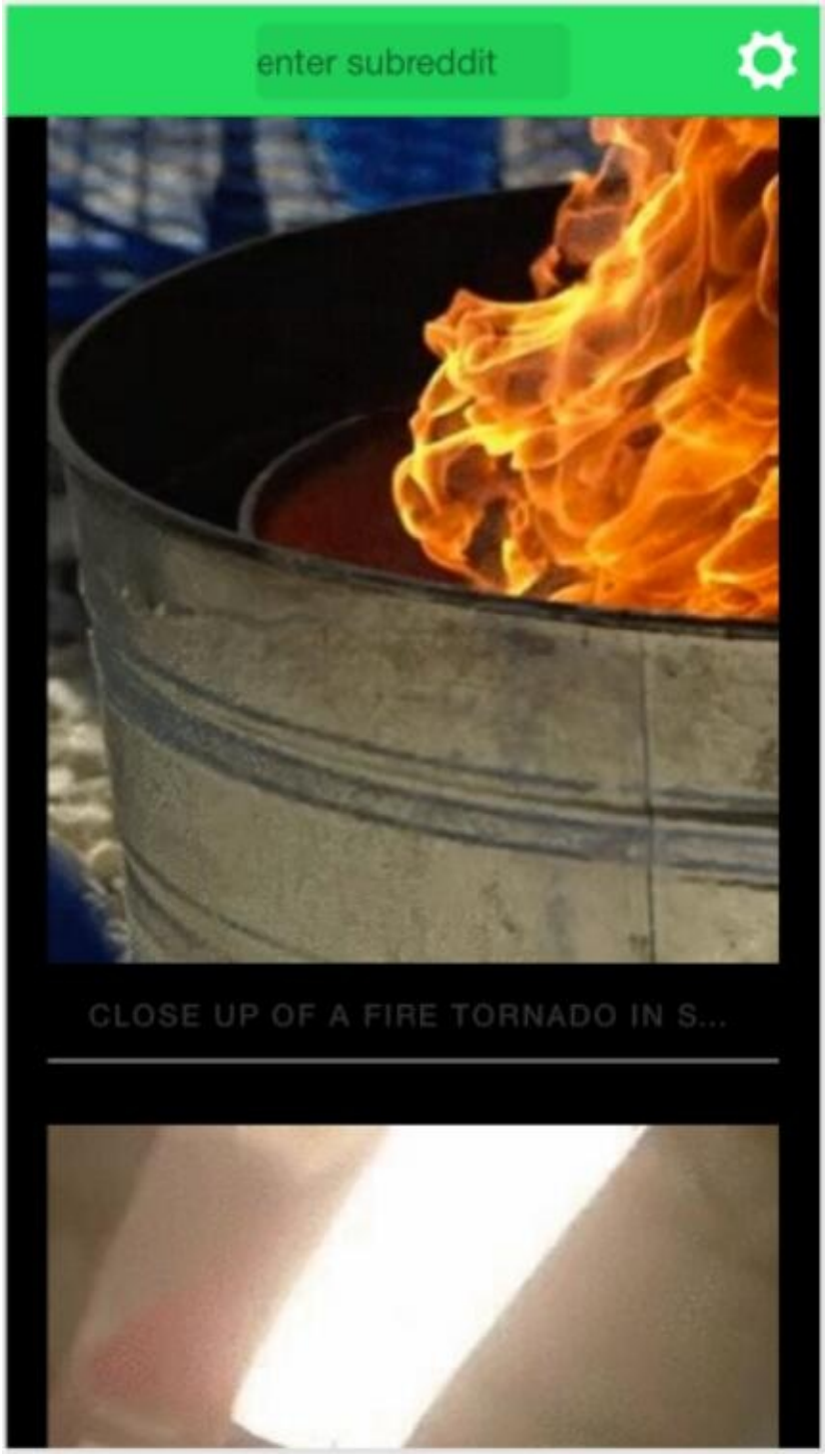
> 修改 **src/pages/home/home.html** 为如下：

```
<ion-header>
    <ion-navbar color="secondary">
    <ion-title>
    <ion-searchbar color="primary" placeholder="enter subreddit name..."  [(ngModel)]=
"subredditValue" [formControl]="subredditControl" value=""></ion-searchbar>
    </ion-title>
    <ion-buttons end>
    <button ion-button icon-only (click)="openSettings()"><ion-icon name="settings"></
ion-icon></button>
    </ion-buttons>
    </ion-navbar>
</ion-header>
<ion-content>
    <ion-list>
        <div *ngFor="let post of redditService.posts">
            <ion-item (click)="playVideo($event, post)" no-lines style="background-col
or: #000;">
            <img src="assets/images/loader.gif" *ngIf="post.showLoader" />
            <video loop [src]="post.data.url" [poster]="post.data.snapshot">
            </video>
            </ion-item>
            <ion-list-header (click)="showComments(post)" style="text-align: left;">

            </ion-list-header>
        </div>
        <ion-item *ngIf="redditService.loading" no-lines style="text-align:center;">
            <img src="assets/images/loader.gif" style="width: 50px" />
        </ion-item>
    </ion-list>
    <button ion-button color="light" full (click)="loadMore()">Load More...</button>
</ion-content>
```

如上所示，我们把帖子的URL作为video元素的源，同时也将预览snapshot作为海报，title作为页首。我们也添加了其他一些东西。
我们添加了一个点击处理器当视频被点击的时候调用*playVideo*，传入*$event*（稍后解释）以及点击到的post的引用。对于有一个单独的点击事件用于在InAppBrowser中启动这个主题。
我们也给列表添加了加载GIF动画。当用户点击视频的时候，需要一点时间来加载他，所以我们加上一个加载动画这样用户知道应用在处理一些事情。没有他的话，用户可能会觉得应用啥都没干。
我们现在来给*loadSettings*函数加点代码，这样运行应用的时候就会调用*fetchData*（因为*loadSettings*是在构造器中调用的）。这就是应用会发生改变的做法，我们稍后来改动他。
> 修改 **src/pages/home/home.ts** 的 **loadSettings** 函数为如下：

```
loadSettings(): void {
    this.redditService.fetchData();
}
```

如果在浏览器中重新加载应用的话，应该可以看到这样的画面。



看起来还是蛮矬的，但是还是稍有改进，因为我们可以看到一些酷的GIF展示出来（现在还不能播放）。我们再改改。

播放**GIF**（视频）

现在列表里面GIF准备就绪，我们现在要让他们摇摆起来。我们所以现在来实现*playVideo*函数。

> 修改 **src/pages/home/home.ts** 的 **playVideo** 函数为如下：

```
playVideo(e, post): void {
    //创建视频的引用
    let video = e.target.getElementsByTagName('video')[0];
    if(!post.alreadyLoaded){
        post.showLoader = true;
    }
    //切换视频播放
    if(video.paused){
        //展示加载gif
        video.play();
        //一旦开始播放视频，隐藏加载gif
        video.addEventListener("playing", function(e){
            post.showLoader = false;
            post.alreadyLoaded = true;
        });
    } else {
        video.pause();
    }
}
```

用户点击视频的时候，我们会将视频在播放和暂停之间切换。我们用模板传入的事件来获取视频本身，然后就可以对这个视频进行播放或者暂停。此处我们也切换了*showLoader*属性以决定加载图标显示与否。我们只需要用户在第一次点击视频的时候显示加载动画，其他时间只在用户暂停的时候显示，所以在切换他之前我们先检查一下*alreadyLoaded*标记。

在In App Browser里启动Comments

还记得我们设置应用的时候，有安装In App Browser插件吧？我们要用上了。当用户点击视频的头的时候，我们会启动一个浏览器来展示原帖。

> 添加以下导入语句到 **src/pages/home/home.ts** 顶部：

```
import { InAppBrowser } from 'ionic-native';
```

> 修改 **src/pages/home/home.ts** 的 **showComments** 函数为如下：

```
showComments(post): void {
    let browser = new InAppBrowser('http://reddit.com' + post.data.permalink,'_system'
);
}
```

跟其他函数对比，这个函数很简单了。我们简单的获取了帖子数据的连接燃油使用InAppBrowser来启动这个链接。

加载更多GIF

现在还有个重要的函数没有实现：*loadMore*，这个是用户点击'LoadMore'按钮的时候调用的函数。这个函数要做的就是加载下一页的GIF。由于我们设置好了*fetchData*函数，所以这个功能就非常简单了。

> 添加以下函数到 **src/providers/reddit.ts** ：

```
nextPage(){
    this.page++;
    this.fetchData();
}
```

> 修改 **src/pages/home/home.ts** 的 **loadMore** 函数为如下：

```
loadMore(): void {
    this.redditService.nextPage();
}
```

我们所作的只是增加页数然后重新调用*fetchData*，简单！！！

更换Subreddit

最后需要完成的函数是*changeSubreddit*（后面还有一点点）函数。但是，首先，我们的给**Reddit**提供者添加一个新的函数来操作subreddit的变更。

> 添加以下函数到 **src/providers/reddit.ts** 顶部：

```
resetPosts(){
    this.page = 1;
    this.posts = [];
    this.after = null;
    this.fetchData();
}
```

> 修改 **src/pages/home/home.ts** 的 **changeSubreddit** 函数为如下：

```
changeSubreddit(): void {
    this.redditService.resetPosts();
}
```

我们早先把最难的骨头啃下来，这也是另一个非常简单的函数。当subreddit改变的时候我们需要重置所有相关事物。如果我们已经来到了‘chemicalreactiongifs’第五页的时候，我们在切换到‘perfectloops’之后就不用继续去加载第五页了。我们这里所作的就是重置页面，清理帖子数据，清理after值，然后重新调用*fetchData*函数。我们已经在他处设置了subreddit，所以调用*fetchData*的时候他会直接使用当前的subreddit。

总结

本课所讲内容是整个应用的大部分了，还有些许待作工作，但是现在可以稍作歇息。此时你的应用已经可以很好的工作了 -- 看起来还是很难看并且没有保存设置的能力，但是他能正常跑起来。下节课就来解决这两个问题。

第五课：设置

现在应用运行良好 -- 来自reddit的GIF，过滤了文件格式，然后以*video*元素进行展示。
我们当前设置了**gifs**作为默认subdreddit，对于本应用来讲是个不错的选择，用户也可以根据喜好对他进行变更。当你离开应用稍后重新进入的时候，他会重新设置为默认的**gifs**subreddit。当用户对**gifs**不敢兴趣的时候这会变得很烦人。
所以我们将要做的事制作一个**Settings**页，用户可以在其中进行如下操作：

- 默认subreddit
- 每页加载的GIF数量
- 排序方式

实现类似这样的功能也让我们有机会去学习如何永久存储数据这个应用将在用户返回的时候记得用户设置。

制作一个**Settings**页

我们将使用**Modal**模态页来打开设置页。打开模态页和普通页面有些许不同，普通页面的打开基本就是应用的导航流的操作了。模态页基本上是一个可以在页面上显示然后关闭的单独页面（你应该对其他网页模态页很熟悉，例如Facebook的照片预览）。
任何页面都可以作为模态页来打开，所以我们可以像创建其他页面那样来创建我们的Settings页面。我们先从定义Settings页的模板开始。
> 修改 **src/pages/settings/settings.html** 为如下：

```
<ion-header>
    <ion-toolbar color="secondary">
        <ion-title>Settings</ion-title>
        <ion-buttons end>
            <button ion-button icon-only (click)="close()"><ion-icon name="close"></io
n-icon></button>
        </ion-buttons>
    </ion-toolbar>
</ion-header>
<ion-content padding>
    <ion-card>
        <ion-card-header>
        About
        </ion-card-header>
        <ion-card-content>
        <p><strong>Giflist</strong> is a client for <strong>reddit</strong>
        that will endlessly <strong>stream GIFs</strong> from <strong>any
        subreddit that predominantly contains gifs</strong>. By default
        it uses the popular <strong>gifs</strong> subreddit, but you can
```

```
        change this to whatever you like, e.g: perfectloops, me_irl,
        chemicalreactiongifs.</p>
        <br />
        <p>To play a GIF, just tap it. To view the original submission for
        any GIF, just tap the title section and it will open in reddit.
        You can configure some settings for the application below.
        Enjoy!</p>
        </ion-card-content>
    </ion-card>
    <h3>Subreddit</h3>
    <ion-input type="text" placeholder="enter subreddit name..." [(ngModel)]="subreddi
t"></ion-input>
    <h3>Sort</h3>
    <ion-segment color="secondary" [(ngModel)]="sort">
    <ion-segment-button value="hot">
    Hot
    </ion-segment-button>
    <ion-segment-button value="top">
    Top
    </ion-segment-button>
    <ion-segment-button value="new">
    New
    </ion-segment-button>
    </ion-segment>
    <h3>Per Page</h3>
    <ion-segment color="secondary" [(ngModel)]="perPage">
    <ion-segment-button value="5">
    5
    </ion-segment-button>
    <ion-segment-button value="10">
    10
    </ion-segment-button>
    <ion-segment-button value="15">
    15
    </ion-segment-button>
    </ion-segment>
    <button ion-button full (click)="save()" color="secondary" style="margin:20px 0px;
">Save Settings</button>
</ion-content>
```

这里没做什么疯狂的事情。我们像往常那样定义了navbar，然后给他添加了一个按钮用于调用*close*函数关闭窗口。
我们添加了一个用来向用户展示应用信息，然后我们定义了一些不同的输入域。我们定义了一个文本输入域来设置subreddit，我们使用了片段组件来控制**sort**和**perPage**值。所有这些数据都通过*[(ngModel)]*进行了双向数据绑定，这样在类定义里面就可以很容易访问到了。
> 修改 **src/pages/settings/settings.ts** 为如下：

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

@Component({
    selector: 'page-settings',
    templateUrl: 'settings.html'
})

export class SettingsPage {
    perPage: number;
    sort: string;
    subreddit: string;

    constructor(public view: ViewController, public navParams: NavParams) {
        this.perPage = this.navParams.get('perPage');
        this.sort = this.navParams.get('sort');
        this.subreddit = this.navParams.get('subreddit');
    }

    save(): void {
        let settings = {
            perPage: this.perPage,
            sort: this.sort,
            subreddit: this.subreddit
        };
        this.view.dismiss(settings);
    }

    close(): void {
        this.view.dismiss();
    }
}
```

可能你一眼就看到了我们在这里用**navParams**来获取一些值。早Home页启动Settings页的时候我们会传过来一些值作为Settings页的预设值。
另一个比较奇怪的是我们导入了**ViewController** -- 当以Modal的方式来启动本页的时候我们需要用到他在页面里关闭本身。在这里我们需要在用户点击关闭或者保存按钮的时候关闭页面。如果用户点击关闭按钮的话，我们就不用保存设置。
在接下来的部分我们会处理设置的保存，但是你可以会想要怎么才能做到呢 -- 这里的save函数不是用来保存一些数据的吗？由于Modal的工作方式的原因，我们可以从以模态框方式启动的页面在关闭的时候返回数据，所以在这里我们调用视图的*dismiss*函数的时候将会把新的设置值回传到Home页面，也就是Setting页面的启动也 -- 我们会在其中操作数据的保存。

以模态页的方式打开**Settings**页

现在，定义好了Settings页面，我们需要已模态也的方式启动他。我们早就在页首设置了一个按钮来打开设置页，所以我们只要完成*openSettings*函数就可以了。
> 修改 **src/pages/home/home.ts** 的 **openSettings** 函数为如下：

```
openSettings(): void {
    let settingsModal = this.modalCtrl.create(SettingsPage, {
        perPage: this.redditService.perPage,
        sort: this.redditService.sort,
        subreddit: this.redditService.subreddit
    });

    settingsModal.onDidDismiss(settings => {
        if(settings){
            this.redditService.perPage = settings.perPage;
            this.redditService.sort = settings.sort;
            this.redditService.subreddit = settings.subreddit;
            //this.dataService.save(settings);
            this.changeSubreddit();
        }
    });

    settingsModal.present();
}
```

你也看到了，我们将Settings页传入了正在创建的Modal，同时也传入了Settings需要获取的数据。
我们设置了*onDisDismiss*监听器用于侦测Moal关闭的时候回传的数据。由于我们只在用户点击保存的时候才会穿数据，所以代码也只会在此时机运行。如果回传了设置数据，我们就要用新的值来更新当前值，同时我们也要使用*dataService*来保存值到存储（我们还是需要去定义这个服务）。由于我们现在有了新的设置，所以我们需要调用*changeSubreddit*来重置所有事情和根据新设置加载新的GIF。
注意：由于我们目前没有实现数据服务，我们目前会注释掉这个服务的调用。
最后，我们调用*present*方法来将模态框展示给用户。

保存数据

谜团的最后一层是制作Data服务来保存设置到存储以及从存储获取设置。
我们将使用Ionic提供的**Storage**服务来存储数据。这也是我最喜欢的Ionic 2新增功能之一，因为他让复杂的存储流程变得超级简单。存储数据中问题有很多，例如操作系统随机清理本地缓存数据，但是这个存储服务自动使用最佳的存储方式。
> 修改 **src/providers/data.ts** 为如下：

```
import { Injectable } from '@angular/core';
import { Storage } from '@ionic/storage';

@Injectable()
export class Data {
    constructor(public storage: Storage) {
    }

    getData(): Promise<any> {
        return this.storage.get('settings');
    }

    save(data): void {
        let newData = JSON.stringify(data);
        this.storage.set('settings', newData);
    }
}
```

首先，我们在构造器里注入了存储服务。然后我们有两个函数，一个用于获取数据，一个用于存储数据。*getData()*函数将返回所有存储与存储空间中的'settings'上的数据，*save()*将把新的数据存放到存储空间的'settings'字段上。我们以及在Settings模态框关闭的时候调用了*save()*，现在我们只需要定义*localSettings*函数用于在应用打开的时候加载之前保存的设置。
> 修改 **src/pages/home/home.ts** 的**loadSettings**函数为如下：

```
loadSettings(): void {
    this.dataService.getData().then((settings) => {
        if(settings && typeof(settings) != "undefined"){
            let newSettings = JSON.parse(settings);
            this.redditService.settings = newSettings;
            if(newSettings.length != 0){
                this.redditService.sort = newSettings.sort;
                this.redditService.perPage = newSettings.perPage;
                this.redditService.subreddit = newSettings.subreddit;
            }
        }
        this.changeSubreddit();
    });
}
```

此处我们简单的调用了data服务的*getData*函数，当数据返回的时候，我们将他JSON为一个对象，读取数据到**Reddit**提供者的成员变量。然后我们调用了*changeSubreddit*函数来触发使用新数据加载新帖子。
现在我们有了数据服务的实现，我们可以拿掉拿掉之前数据解除之前的注释。
> 修改 **home.ts**的 **openSettings** 函数为如下：

```
openSettings(): void {
    let settingsModal = this.modalCtrl.create(SettingsPage, {
        perPage: this.redditService.perPage,
        sort: this.redditService.sort,
        subreddit: this.redditService.subreddit
    });

    settingsModal.onDidDismiss(settings => {
        if(settings){
            this.redditService.perPage = settings.perPage;
            this.redditService.sort = settings.sort;
            this.redditService.subreddit = settings.subreddit;
            this.dataService.save(settings);
            this.changeSubreddit();
        }
    });

    settingsModal.present();
}
```

总结

现在我们完成了设置页和数据服务，应用的功能就全部完成了！虽然现在看起来还是那么难看。
所以我们还有一节课的时间来添加一些自定义的样式让他好看些。

第六课：自定义样式

我们快要完成这个应用了，做完一点点自定义样式之后，就算正式完成了。在定义我们的样式之前我们将要给模板文件添加一些类。
> 修改**src/pages/home/home.ts** 为如下：

```
<ion-header>
    <ion-navbar color="secondary">
        <ion-title>
        <ion-searchbar color="primary" placeholder="enter subreddit name..."   [(ngMod
el)]="subredditValue" [formControl]="subredditControl"  value=""></ion-searchbar>
        </ion-title>
        <ion-buttons end>
        <button ion-button icon-only (click)="openSettings()"><ion-icon   name="settin
gs"></ion-icon></button>
        </ion-buttons>
    </ion-navbar>
</ion-header>
<ion-content>
<ion-list>
    <div *ngFor="let post of redditService.posts">
    <ion-item (click)="playVideo($event, post)" no-lines  style="background-color: #00
0;">
    <img src="assets/images/loader.gif" *ngIf="post.showLoader" class="video-loader" />

    <video loop [src]="post.data.url" [poster]="post.data.snapshot">
    </video>
    </ion-item>
    <ion-list-header (click)="showComments(post)" style="text-align: left;">
    {{post.data.title}}
    </ion-list-header>
    </div>
    <ion-item *ngIf="redditService.loading" no-lines style="text-align:center;">
    <img src="assets/images/loader.gif" style="width: 50px" />
    </ion-item>
    </ion-list>
    <button ion-button color="light" full (click)="loadMore()">Load More...</button>
</ion-content>
```

如果你复习过基础部分的话，你应该知道有一些定义样式的方法，如果你跳过了基础部分的定制主题的话，强烈建议你回去读一遍。 **>** 修改**src/pages/home/home.scss** 为如下：

```
page-home {

    ion-item {
        background-color: #000;
        position: relative;
    }

    ion-item .video-loader {
        position: absolute;
        top: 10px;
        right: 10px;
        width: 25px;
        height: 25px;
    }

    ion-list-header {
        background-color: #fff;
    }

    ion-label {
        margin: 0px !important;
    }

    ion-list {
        margin: 0px !important;
    }

    ion-item-content, .item-inner, .item {
        margin: 0 !important;
        padding: 0 !important;
        text-align: center;
    }

    video {
        max-width: 100%;
        height: auto;
        background-color: #000;
    }
}
```

这些都是很基本的样式，大部分我们都只是改变一下颜色，移除间隙和边距。这里最重要的样式是*ion-item-content*和*video*样式用于缩放视频以适配离别。理想情况下我们希望视频从宽度上填满整个项，但是本案例中GIF不会都有肖像，所以我们让GIF居中显示，然后添加黑色背景，这样其实也蛮好看的。

现在我们将针对iOS做一些特别的样式。我们顶部的搜索条看起来有点小，因为我们把他加入到部分，所以我们得把他变大点。搜索条在Android的显示很好，所以我们不想要样式在Android上产生作用。

> 添加如下样式到 **src/app/app.scss**：

```
.ios {
    ion-title {
        padding: 0px 35px 1px 0px;
    }
}
```

当在iOS上运行的时候，Ionic将自动添加一个ios类到*<_body>_*这样我们可以通过以上方法对iOS进行定制。同理，也可以通过*md*对Android进行定制，*wp*对Windows进行定制。接下来，我们要给body标签添加一个背景色，这个我们将会在同一的*.scss*文件中添加。
同时我们也会对整个应用进行样式定制。
> 将以下样式添加到 **src/app/app.scss**：

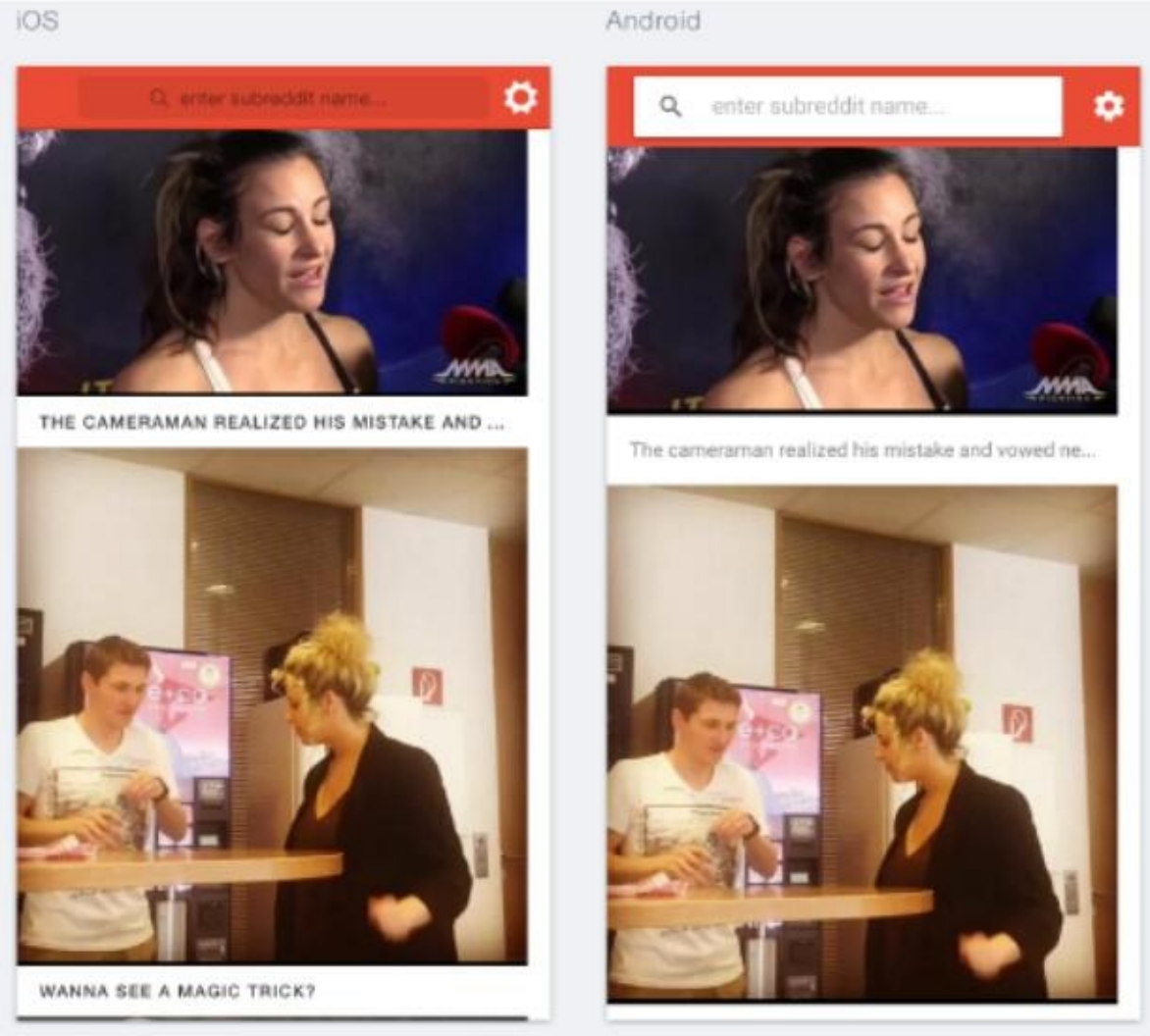
```
body {
    background-color: #e74c3c;
}
```

终于，我们来到了定义应用共享变量的地方了，**variables.scss**文件。
> 修改 **src/theme/variables.scss**到如下样式：

```
$colors: (
    primary: #ecf0f1,
    secondary: #e74c3c,
    danger: #f53d3d,
    light: #e74c3c,
    dark: #222222
);

$searchbar-ios-background-color: #e74c3c;
$searchbar-ios-input-background-color: #c94234;

$button-ios-border-radius: 0px;
```

此处我们进行了一些颜色变更，对搜索条进行了一些修改，设置了iOS的按钮边缘半径让我们的‘Load More’从矩形变成了圆角矩形。

好看多了！如果你查看Android版本的话（在Chrome Dev Tools中设置Emulator为Samsung或者其他Android设备，或者使用*ionic server -l*命令），效果是这样的：



（请忽略右边的白色条，他不会出现在真实设备上的）
他看起来明显不同了，但是看起来还是很不错。Ionic最好的事情之一是他自动根据运行平台进行调整适配。通常，越少跟平台纠缠越好。如果你主要是用iOS设备的话，或者主用Android设备，其他平台的样式看起来可能会有点怪异，但是此平台用户而言可能不会那么怪异。如果有很好的理由，否则不应该去改动平台特定样式（当然可以实现的）来让他看起来跟另一平台相同。

总结

跟其他课程相比本节课相当简短，但是同时他也给你展示了付出一点点的努力为应用制作一个不错的自定义主题是多么的简单。

结论

恭喜你完成了Giflist指南。通过开发这个应用，我们学习到了很多东西，主要的有：

- 如何通过第三方API获取数据
- 如何存储和获取本地数据
- 如何使用Observable
- 如何在模板中使用条件语句
- 如何使用事件

改进的空间永远都存在，特别是当你学习事物的时候。遵循指导手册固然很好，但是自己去学习弄清楚一些事情就更完美了。希望你有足够的背景知识来自己完成一些功能扩展，以下是一些想法：

- 为应用创建自己的自定义主题【简单】
- 创建一个‘Random’按钮来从预定义数组中获取随机的subreddit【简单】
- 将应用从返回GIF修改为返回图片【中等】
- 将应用从修改为返回图片和GIF【困难】
- 允许用户保存喜欢的GIF到自己的列表这样可以通过在搜索条里面输入‘favorite’来浏览他们【非常难】

记住，在你学习的过程中，Ionic 2文档是你最好的朋友。

接下来？

我们已经有一个完成的应用了，但是这不是故事的结局。你需要让他运行到真实设备上去，提交到应用商店，这不是个简单的任务。本书的最后部分会带你完成这些内容。你可以先看看。

第五章

每日一拍

第一课：介绍

Snapaday是基于大家都喜欢的一个本地插件相机的。这个论点实际上我没什么理论数据支持，但是相机绝对是最酷的整合之一。这也是人们经常做斗争的插件之一，他就是那种表面看起来很简单，内部却各种手法的东西。
实际上，Snapaday可以说是本书的“本地插件应用” -- 在介绍如何使用相机的同时，我们也会介绍整合本地通知和社交分享（我们得把这些自拍分享到Facebook，对吧？）。

关于Snapaday

Snapaday实际上是我的Mobile Development for Web Developers课程中的Ionic 1范例，所以在我考虑从Ionic 1转向Ionic 2的时候，这个一个绝佳的例子 -- 特别是那些已经实现过Ionic 1版的Snapaday的人们。
主要想法是用户每天使用应用拍照，可以通过滑动展示自己每天的改变（看过这个视频吗？）。为了更精确的表达出来，应用的实际功能如下：

- 允许用户每天拍照一张（只允许每天一张）
- 在一个列表中展示用户的所有照片列表
- 允许用户删除不想要的照片
- 以快速滑动的方式回放照片
- 分享照片
- 通过本地通知进行提醒

在制作过程中需要学习的一些概念：

- 如何整合本地插件
- 如何使用Camera API
- 如何使用File API
- 如何使用本地通知
- 如何使用模态框
- 如何制作一个自定义提佛那个这
- 如何永久存储数据

我们先来看看最终效果图：

课程结构

1. 准备工作
2. 布局
3. 使用Camera拍照
4. 存储和获取相片
5. 自作一个自定义管道和所有相片的飞页

6. 整合Local Notification
7. 自定义样式

准备好了吗？
现在你知道了你要做什么，那么我们就可以开始了。

第二课：准备工作

本课是在旅程继续进行之前的一些准备工作。我们要生成应用，设置所有组件和需要用到的Cordova插件。完成本课之后我们应该有一个万事俱备的项目骨架，可以接着进行编码工作。开始新项目的第一准则是确保使用的是最新版的Ionic和Cordova，如果最近没有更新过的话可以运行如下命令：

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

或者

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

如果在安装Ionic或者生产新项目的时候遇到任何问题的话，检查下是否安装了最新版的Node。安装完之后，再次安装ionic之前请运行：

```
npm uninstall -g ionic npm cache clean
```

生成新应用

本应用将使用空白初始模板，跟名字说的一样，就是个空的Ionic项目。但是会有一个内置的页面名为**home**，我们下节课中将用作列表显示页。
> 运行如下命令生成新项目：

```
ionic start snapaday blank --v2
```

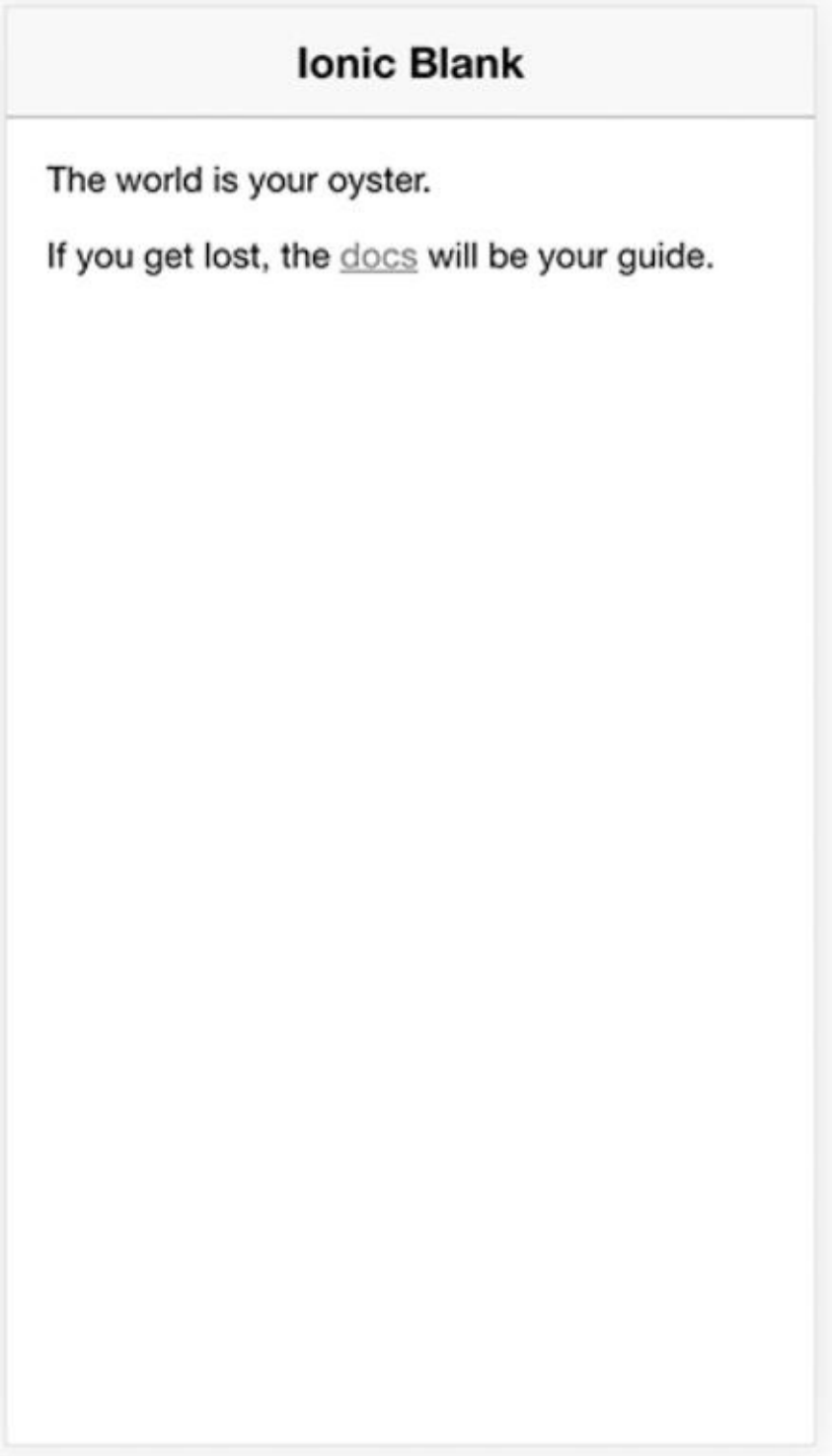
> 运行如下命令将新生成的项目作为当前目录：

```
cd snapaday
```

现在可以在你中意的编辑器中打开这个项目了。通过以下命令可以预览创建的应用：

```
ionic serve
```

看起来是这样的：



创建需要的组件

应用的架构很简单，我们会有两个页面：主页含有照片和一个展示页。我们已经有了一个**home**组件用作列表页，所以我们只要创建一个**Slideshow**页面就可以了。
> 运行如下命令生成 **Slideshow** 页面：

```
ionic g page Slideshow
```

创建需求的服务

跟我们的两个页面一样，我们需要创建一个数据服务来存储和获取照片，由于我们在应用中有很多警告信息，所以我们也需要创建一个服务来更方便的处理他们。
> 运行如下命令生成 **Data** 提供者：

```
ionic g provider Data
```

> 运行如下命令生成 **Alert** 提供者：

```
ionic g provider SimpleAlert
```

我们还需要创建自定义的Pipe来讲拍照日期转换成更友好的内容，例如‘5 天前’。 **>** 运行如下命令生成 **DaysAgo** 管道：

```
ionic g pipe DaysAgo
```

创建数据模型

我们将为我们的照片创建一个数据模型用于更简单的创建和更新他们。不幸的是，没有现有命令用来生成，所以我们要手动创建。
> 在 **src** 文件夹内创建一个新的文件夹名为**models**：
> 在 **models** 文件夹内创建一个新的文件名为 **photo-model.ts**：

在App Module中添加页面和服务

为了能够在项目里面可以使用这些页面和服务，我们需要将它们添加到**app.module.ts**文件里。所有我们自己创建的页面都需要添加到*declarations*数组和*entryComponents*数组里，所有我们创建的数据提供者都需要添加到*providers*数组，其他自定义组件或者管道（pipe）只需要添加到*declarations*数组即可。我们的数据模型只是一个简单的类，我们需要在任何地方使用，所以不用在模组里面设置。

> 修改**src/app/app.module.ts**到以下：

```
import { NgModule } from '@angular/core';
import { IonicApp, IonicModule } from 'ionic-angular';
import { MyApp } from './app.component';
import { Storage } from '@ionic/storage';
import { HomePage } from '../pages/home/home';
import { DaysAgo } from '../pipes/days-ago';
import { SlideshowPage } from '../pages/slideshow/slideshow';
import { SimpleAlert } from '../providers/simple-alert';
import { Data } from '../providers/data';

@NgModule({
    declarations: [
        MyApp,
        HomePage,
        SlideshowPage,
        DaysAgo
    ],
    imports: [
        IonicModule.forRoot(MyApp)
    ],
    bootstrap: [IonicApp],
    entryComponents: [
        MyApp,
        HomePage,
        SlideshowPage
    ],
    providers: [Storage, Data, SimpleAlert]
})
export class AppModule {}
```

注意，我们除了自己创建的*Data*提供者之外，我们还加入了一个*Storage*。Storage是Ionic提供的，可以通过它保存和获取数据 -- 我们后续会用到。

添加需要的平台

在给指定平台制作应用之前，你需要将它们添加到你的项目。
> 运行以下命令添加**iOS**平台：

```
ionic platform add ios
```

> 运行以下命令添加**Android**平台：

```
ionic platform add android
```

添加需要的Cordova插件

这个应用将会用到不同的Cordova插件。记住，Cordova插件只能在真实设备上运行。我将在添加他们的时候解释。
> 运行以下命令添加**SQLite**插件：

```
ionic plugin add cordova-sqlite-storage
```

这个插件让你可以访问本地存储SQLite数据库。我们在此应用中添加他的原因是Ionic本地存储服务可以使用插件提供的稳定输出存储。
> 运行以下命令添加本地通知插件：

```
ionic plugin add de.appplant.cordova.plugin.local-notification
```

这个插件允许我们为应用创建本地通知。和推送通知不一样，本地通知是完全在用户设备上处理的，不需要外部服务。
> 运行以下命令添加**Camera**插件：

```
ionic plugin add cordova-plugin-camera
```

这个插件给你访问用户相机和返回照片的能力。提供相机访问权限的同时，也允许你从用户照片库中获取照片。
> 运行以下命令添加**File**插件：

```
ionic plugin add cordova-plugin-file
```

File插件允许你和设备文件系统交互，我们将用来移动应用来照的照片到其他地方。
> 运行以下命令添加**Social Sharing**插件：

```
ionic plugin add cordova-plugin-x-socialsharing
```

社交分享插件是一个通用插件，允许用户分享到大量不同平台（像是社交媒体，邮件，SMS等等）或者是触发特定平台的分享。
> 运行以下命令添加**Status Bar**插件：

```
ionic plugin add cordova-plugin-statusbar
```

我们给所有项目添加此插件用来在应用中控制状态栏（设备屏幕顶部的状态条，包括时间，电池信息等等）。
> 运行以下命令添加**Splash Screen**插件：

```
ionic plugin add cordova-plugin-splashscreen
```

此插件允许我们控制闪屏（打开应用的时候的全屏画面）。
> 运行以下命令添加**Keyboard**插件：

```
ionic plugin add ionic-plugin-keyboard
```

这个插件允许我们控制软键盘。
> 运行以下命令添加**Whitelist**插件：

```
ionic plugin add cordova-plugin-whitelist
```

所有应用会用到这个插件，他定义了应用里可以加载什么样的资源。没有他的话，你尝试加载的资源都会不成功。
添加了这个插件后，你也需要到**index.html**中定一个一个“Content Security Policy”。我们将添加一个非常宽松的策略实际上允许我们加载任何资源。基于你的应用，你可以提供一个更严格的策略，但是对于开发而言开放性策略就可以了。
> 修改 **src/index.html**文件，添加一下**meta**标签：

```
<meta http-equiv="Content-Security-Policy" content="font-src 'self' data:;
img-src * data:; default-src gap://ready file://* *; script-src 'self'
'unsafe-inline' 'unsafe-eval' * ; style-src 'self' 'unsafe-inline' *">
```

> 运行以下命令添加**Crosswalk**插件：

```
ionic plugin add cordova-plugin-crosswalk-webview
```

这个另一个每个应用都要添加的插件，但是你也可以先不添加。添加了这个插件后，在你编译Android的时候将会使用“Crosswalk”。Android有很多问题，特别是老设备，因为有太多不同的团建版本，不同的版本有不同的浏览器（记住，鉴于我们是制作HTML5应用，他实际上就是一个搭载的浏览器用来运行我们的应用）。Crosswalk做的是将一个现代的浏览器打包到应用中，这样一来应用无论是运行在什么设备上，都会使用相同的浏览器来运行，并且Crosswalk浏览器可以很好改善执行效率。
唯一的不足之处就是你的应用尺寸明显的变大了很多。总体上，我觉得这很值得，我也建议你使用他，如果你接受不了的话，也可以不用。更多关于Crosswalk Project的信息，请参考网站：https://crosswalk-project.org/

设置图片

制作此应用的时候，会用到一些图片。你下载的包里面已经包含了这些图片，但是你需要去生成的项目里面设置好他们。
> 将下载包 **src/assets**文件夹下面的**images**文件夹复制到应用里的 **src/assets**下面

总结

就这样！我们设置好了，准备好继续前几，现在我们开始进入到有趣的部分了。

第三课：布局

我发现制作一个应用最好从基础布局开始 -- 他基本可以看作是线图联系，帮助固化应用的需求。我们这个应用的布局没啥特别的东西，但是有一个需要添加的用户界面元素需要一点点技巧。
我们这个应用有两个页面，主页和滑页，本课中将会制作他们的模板。滑页非常简单，所以本课的大部分都是关于主页的。

Home页

主页有一个用户拍的照片的列表，用户可以在其中删除它们，一个拍新的照片的选项（只有在当日没拍的情况下），一个展示图片滑页的选项。效果图上这样的：

这个布局最有技巧性的是顶部的“SMILE!”图片，实际上是一个按钮，用于拍照的按钮，同时，他也只会在用户当日没有拍照的情况下才显示。如果用户当日没有拍照的话就不会显示的。

我们先从基础布局入手，然后看怎么实现拍照按钮。

> 修改**src/pages/home/home.html** 为如下：

```
<ion-header>
<ion-navbar color="danger">
<ion-title>
<img src="assets/images/logo.png" />
</ion-title>
<ion-buttons end>
<button ion-button icon-only (click)="playSlideshow()"><ion-icon name="play"></ion-icon
></button>
</ion-buttons>
</ion-navbar>
</ion-header>
<ion-content>
<ion-list>
<ion-item-sliding>
<ion-item>
<img src="http://placehold.it/100x100" />
<ion-badge item-right light>0 days ago</ion-badge>
</ion-item>
<ion-item-options>
<button ion-button icon-only light (click)="removePhoto(photo)"><ion-icon name="trash">
</ion-icon></button>
</ion-item-options>
</ion-item-sliding>
</ion-list>
</ion-content>
```

我们把这个模板打碎来讲解。模板的第一个部分是导航条：

```
<ion-navbar color="danger">
<ion-title>
<img src="assets/images/logo.png" />
</ion-title>
<ion-buttons end>
<button ion-button icon-only (click)="playSlideshow()"><ion-icon name="play"></ion-icon
></button>
</ion-buttons>
</ion-navbar>
```

ion-navbar 允许我们添加一个页首条到我们的应用顶部用于持有按钮，标题，甚至在需要的时候直接整合Ionic导航系统来展示一个回退按钮。
我们给navbar添加了*danger*属性将他样式化到我们的*danger*颜色，这个颜色变了色在**variables.scss**中定义的（后续会配置）。
在navbar中我们用到了*ion-title*，基本就是用来展示当前页面的文本标题和logo的。同时我们也用到了*ion-buttons*来创建按钮。通过指定*end*属性，所有按钮将被排列在右边，当我们指定的是'start'属性的时候，将会从左边开始排列。

最后，我们将所有按钮列入*ion-buttons*中。这个按钮用到了一个play图标，有一个点击处理器，处理器将调用**home.ts**文件里的*playSlideshow()*函数（这个函数当前未创建）。
模板接下来的部分是内容部分，包括了一个列表：

```
<ion-content>
<ion-list>
<ion-item-sliding>
<ion-item>
<img src="http://placehold.it/100x100" />
<ion-badge item-right light>0 days ago</ion-badge>
</ion-item>
<ion-item-options>
<button ion-button icon-only color="light" (click)="removePhoto(photo)"><ion-icon name=
"trash"></ion-icon></button>
</ion-item-options>
</ion-item-sliding>
</ion-list>
</ion-content>
```

在看列表之前，我们看到所有东西都被包围在*ion-content*里 -- 这就是页面主体内容展示的地方，在大部分情况下除了导航条外的所有内容都要放在这里。
跟一个普通列表其实也差不多，看：

```
<ul>
    <li></li>
    <li></li>
    <li></li>
</ul>
```

Ionic里面创建列表的方式基本一样：

```
<ion-list>
    <ion-item></ion-item>
    <ion-item></ion-item>
    <ion-item></ion-item>
</ion-list>
```

当然，我们的看起来稍微复杂点，所以我们还是来研究研究。首先与常理不一样的是我们在*ion-list*加入了*no-lines*属性。就像给navbar添加的danger属性一样，这个属性会对我们的列表起作用，他会使列表项不显示边缘。
接下来的变得多一点技巧性，也就是我们设置滑动项的地方。一个，和普通的不同，他有两套内容 -- 列表项本身，然后是在用户滑动的时候可见。
第一部分内的代码是普通的定义。项里面是用户拍的照片（我们最后还是会循环所有相片），但是现在我们只是用了个图片占了个位置。我们也在项的右边添加了一个勋章用于展

示这个照片上多久之前拍的。我们现在用的都是假数据，后面再用真数据。
最后，我们来到了代码的第二部分，仅用来展示活动展示的内容。当前应用中，我们只是添加了一个'Delete'按钮，当点击他的时候会传入当前照片的引用。现在传入的*photo*引用不会有实际效果（调用的函数也不会有效果因为都没创建呢），稍后制作相片循环的时候我们在讨论如何建立这个应用。
现在，我们来添加之前讲到的那个条件照片按钮。
> 修改 **src/pages/home/home.html** 的 **content** 部分为如下：

```
<ion-content>
    <ion-list>
        <button ion-item *ngIf="!photoTaken" detail-none (click)="takePhoto()">
            <img src="assets/images/smile.png" />
        </button>
        <ion-item-sliding>
            <ion-item>
                <img src="http://placehold.it/100x100" />
                <ion-badge item-right light>0 days ago</ion-badge>
            </ion-item>
            <ion-item-options>
                <button ion-button icon-only color="light" (click)="removePhoto(photo)"
><ion-icon name="trash"></ion-icon></button>
            </ion-item-options>
        </ion-item-sliding>
    </ion-list>
</ion-content>
```

注意，我们在列表里的滑动选项上面加入了一个新的项，但是我们不是直接用的而是用的*button*然后给他使用*ion-item*指令。在视觉上，这两种方法其实是一样的，但是在移动设备上除了*button*和*a*之外的点击处理都会有一点延迟。我们不想要这个延迟，所以我们使用按钮。由于按钮会默认有自带样式的，而我们这里不想要那些样式，所以我们给他添加了*detail-none*属性。当然，我们也给他添加了一个点击处理器用于触发后面将会创建的*takePhoto*函数，但是这里最重要的还是 nglf 指令。
nglf 是Angular2提供的条件指令之一，允许我们根据数据改变模板显示。在现在的模板中，含有 nglf 指令的元素只会在表达式为*true*的时候才会全部显示。由于我们的表达式是这样的：

```
*ngIf="!photoTaken"
```

所以我们的按钮只有在*photoTaken*为*false*的时候才会显示。稍后，我们会在**home.ts**中创建一个**photoTaken**变量用于追踪用户当日是否拍照。
现在运行以下命令的话：

```
ionic serve
```

会看到如下效果图：

显然，还有很多事情需要去完成，但是目前我们只需要对主页做这么多就够了，所以，我们继续Slideshow 页。

Slideshow 页

就像我之前说过的那样，slideshow页将会超级简单。他就是一个弹出的模态框展示所有用户拍的照片的滑页。我们只需要为照片提供一个容器，一个按钮来重启滑页和一个按钮来关闭页面就可以了。

> 修改 **src/pages/slideshow/slideshow.html** 为如下：

```
<ion-header>
    <ion-toolbar color="danger">
        <ion-title>Play</ion-title>
        <ion-buttons start>
            <button ion-button icon-only (click)="playPhotos()"><ion-icon name="refres
h"></ion-icon></button>
        </ion-buttons>
        <ion-buttons end>
            <button ion-button icon-only (click)="closeModal()"><ion-icon name="close">
</ion-icon></button>
        </ion-buttons>
    </ion-toolbar>
</ion-header>

<ion-content>
    <div class="image-container">
        <img #imagePlayer id="imagePlayer" src = "" />
    </div>
</ion-content>
```

你已经知道navbar和按钮咋工作的了，所以唯一的新玩意是图片播放器（image player）。我们只是在内容显示区域添加了一个图片元素。稍后，我们会添加一些代码循环所有用户的照片，通过改变*src*属性来展示所有照片。我们只要在类里面拿到*img*的引用就可以了，所以我们创建了一个局部变量 #imagePlayer 这样可以直接在类定义中拿到他了。
目前而言，以上就是我们本课的工作量 -- 告诉过你很简单的。

总结

现在，我们应用的两个页面都设置好了，下节课将会进行一些更有趣的事情。实际上，他可能是最有趣的课程之一，因为我们要学习整合本机Camera API然后用他来拍照。

第四课：使用相机拍照

本课的目标是整合Camera这样我们可以用他来拍照了，但是要达到这个目标需要下点功夫。除了要出发相机拍照之外，我们还要：

- 将照片移动到手机上的持久储存
- 在模板中展示这些照片
- 显示其他的照片
- 允许删除照片 这样看来本课程还是蛮大的。在本应用的开始部分我们设置了Ionic Native，也就是基础部分里面的内容，Ionic Native是Ionic包装好的Cordova插件，使用更简单。
  本课程中将用到Ionic Native中的Camera插件和File插件。
  让我们愉快的开始吧！

创建一个**Photo**数据模型

在我们添加相片功能之前，我们需要创建一个数据模型来代表照片对象。当我们想要存放照片存储路径，照片拍摄日期的时候。数据模型就能够很轻松的处理这样的事情，像这样：

```
let photo = new PhotoModel('path/to/image', new Date());
rather than this:
let photo = {
    image: 'path/to/image',
    date: newDate
};
```

区别不是太明显，也不是所有的原因都需要用到，但是是一个很好的模式，特别是当数据越来越复杂的时候。举个复杂数据模型的例子，去看看Quick List课程，如果你还没看过的话。
> 修改 **src/models/photo-model.ts** 为如下：

```
export class PhotoModel {
    constructor(public image: string, public date: Date){

    }
}
```

这个模型很直白，我们传入了图片和日期。如果想要在**home.ts**中使用他的话，得先导入。
> 修改 **src/pages/home/home.ts** 为如下：

```
import { Component } from '@angular/core';
import { NavController } from 'ionic-angular';
import { PhotoModel } from '../../models/photo-model';

@Component({
    selector: 'page-home',
    templateUrl: 'home.html'
})

export class HomePage {
    constructor(public navCtrl: NavController) {
    }
}
```

制作一个简单的Alert服务

Surprise！（译者：惊喜！好像不大好听）在进入开心环节之前，我们还有一个要做的事情。由于很多地方可能会有错误提示（拍照，移动照片），所以我们出发大量的警告框，当然，事情处理好了之后我们也要告诉用户一声不是。
创建警告提示的语法是这样的：

```
let alert = this.alertCtrl.create({
    title: "My Title",
    message: "My Message",
    buttons: [
        {
            text: 'Ok'
        }
    ]
});

alert.present();
```

代码量还是有点的，如果我们在同一个文件里多次调用这个代码，看起来会很乱。所以我们新建另一个服务来处理这样的事情，就像数据模型一样。创建完之后，我们就可以像下面这样去弹出警告框：

```
let alert = this.simpleAlert.createAlert('Oops!', 'Something went wrong.');
alert.present();
```

> 修改 **src/providers/simple-alert.ts** 为如下：

```
import { Injectable } from '@angular/core';
import { AlertController } from 'ionic-angular';

@Injectable()
export class SimpleAlert {

    constructor(public alertCtrl: AlertController){
    }
    createAlert(title: string, message: string): any {
        return this.alertCtrl.create({
            title: title,
            message: message,
            buttons: [
                {
                    text: 'Ok'
                }
            ]
        });
    }
}
```

这里我们导入了AlertController服务用来创建警告框，然后创建了一个函数，结束一个title和一个message，然后返回一个用这些信息组装的的警告框。这虽然给我们创建了警告提示框，但是展示还是要为我们手动去展示出来。

> 修改 **src/pages/home/home.ts** 如下：

```
import { Component } from '@angular/core';
import { NavController } from 'ionic-angular';
import { PhotoModel } from '../../models/photo-model';
import { SimpleAlert } from '../../providers/simple-alert/simple-alert';

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

    }
}
```

注意我们在构造器里面注入了SimpleAlert和NavController，但是没有注入照片数据模型。这是因为每次我们要用到PhotoModel的时候，我们都是通过*new*新建一个实例的，但是对于SimpleAlert而言我们都是一次一次调用他统一实例的同一函数的。

使用**Camera**照相

好了，完成了照片模型和弹窗服务之后 -- 我们终于来到了有趣的部分了，也就是拍照。我们现在要使用Ionic Native的Camera插件，我们得先进行导入： **>** 在 **src/pages/home/home.ts** 中加入以下导入语句：

```
import { Camera, File } from 'ionic-native';
```

现在，我们可以通过**Camera**对象来访问这个功能了。我们将一下如何使用这个插件，但是记住所有Ionic Native里面的插件都可以在这里找到文档：http://ionicframework.com/docs/native/Camera/
在写拍照相关的代码之前，我们将要到构造器里面添加一些变量，这样后面就不要老是导入这个导入那个，我们把应用后面会用到的全部都导入进来好了。我们将要加入一些新的函数，一些引用了其他函数的函数。这样我们就不会遇到函数undefined的问题，我们全部定义好，但是都留空。我们后续会详细实现他们（不全部是在本课中）。
> 修改 **src/pages/home/home.ts** 为如下：

```
import { Component } from '@angular/core';
import { ModalController, AlertController, Platform } from 'ionic-angular';
import { PhotoModel } from '../../models/photo-model';
import { SimpleAlert } from '../../providers/simple-alert';
import { SlideshowPage } from '../slideshow/slideshow';
import { Data } from '../../providers/data'
import { Camera, File } from 'ionic-native';

declare var cordova;

@Component({
    selector: 'page-home',
    templateUrl: 'home.html'
})

export class HomePage {
    loaded: boolean = false;
    photoTaken: boolean = false;
    photos: PhotoModel[] = [];

    constructor(public dataService: Data, public platform: Platform, public  simpleAle
rt: SimpleAlert, public modalCtrl: ModalController, public  alertCtrl: AlertController
) {

    }

    ionViewDidLoad(){
        // Uncomment to use test data
        /*this.photos = [
        new PhotoModel('http://placehold.it/100x100', new Date()),
        new PhotoModel('http://placehold.it/100x100', new Date()),
        new PhotoModel('http://placehold.it/100x100', new Date())
        ]*/
```

```
        this.platform.ready().then(() => {
            this.loadPhotos();
        });

        document.addEventListener('resume', () => {
            if(this.photos.length > 0){
                let today = new Date();
                if(this.photos[0].date.setHours(0,0,0,0) === today.setHours(0,0,0,0)){
                    this.photoTaken = true;
                } else {
                    this.photoTaken = false;
                }
            }
        }, false);
    }

    loadPhotos(): void {
    }

    takePhoto(): any {
    }

    createPhoto(photo): void {
    }

    removePhoto(photo): void {
    }

    playSlideshow(): void {
    }

    sharePhoto(image): void {
    }

    save(): void {
    }
}
```

上面的代码中，我们新增了一个成员变量*loaded*用于保持追踪照片是否都从存储中加载完成（下节课要做的），*photoTaken*用于标记今天是否有拍照，*photos*用于持有所有的照片数据。由于照片只会在应用运行在真实设备的情况下才会加载，所以我们在*ionViewDidLoad*中加入了一些测试数据，我们在此处给*this.photos*添加了一个测试数据这样就可以在浏览器中进行测试。如果你想在测试的时候看到相片呈现，解除其中的注释即可。（记得后面删掉！）
我们已经设置好了所有需要用到的服务的引用，包括稍后要用到的数据服务和平台服务。
在这里我们也添加了一个奇怪的*resume*监听器。resume事件的触发时机是用户把应用发配到后台然后重新使用的时候将会触发。例如，当用户打开了你的应用，然后去玩Facebook，然后有回到我们应用的时候。我们这么做的原因是因为我们的*photoTaken*变量有一些极端的特例。想象一下，某人今天拍照了，但是当他们关闭应用但是没有全部关闭的时候，他只是被发配后台。第二天我们使用这个应用的时候，我们在*loadPhoto*函数中运行的用于判断当日是

否拍照的逻辑将不会执行，因为只是恢复而不是重新加载。所以当应用恢复的时候，我们都要检查最后照片拍摄日期是否跟今日日期椅子，然后根据他去设置*photoTaken*变量。
注意：我们这里加上来*declare var cordova*，这样TypeScript不会抱怨：我根本不知道cordova是啥。
现在我们将实现*takePhoto*函数，这个函数是用来拍照的，所以我们先来个基础版。
> 修改 **takePhoto** 函数为如下：

```
takePhoto(): any {

    if(!this.loaded || this.photoTaken){
        return false;
    }

    if(!this.platform.is('cordova')){
        console.log("You can only take photos on a device!");
        return false;
    }

    let options = {
        quality: 100,
        destinationType: 1, //return a path to the image on the device
        sourceType: 1, //use the camera to grab the image
        encodingType: 0, //return the image in jpeg format
        cameraDirection: 1, //front facing camera
        saveToPhotoAlbum: true //save a copy to the users photo album as well
    };

    Camera.getPicture(options).then((imagePath) => {
        console.log(imagePath);
    },
    (err) => {
        let alert = this.simpleAlert.createAlert('Oops!', 'Something went wrong.');
        alert.present();
    });
}
```

执行Camera代码之前，我们检查了一系列的条件检查。如果之前创建的变量显示数据没有加载完成的话，或者今日已拍照的话，函数内后面的代码就不会执行了。我们也检查了我们是否是运行在‘cordova’平台上，也就是真机上，如果没有的话就直接退出函数。因为啊，你只能在真实设备上访问这些插件，如果不是在真实设备上运行的话，那么除了报错就只能报错了。
接着我们设置了一些选项传入Camera插件，这些配置了我们要干啥，我们要返回啥。这些值可以配置的东西包括，是否要用摄像头或者用户的照片库，返回图片的格式，用前置摄像头呢，还是用后置摄像头等等。我在每个选项后面都添加了注释，来解释他们分别是干啥的。
创建好了选项对象之后，我们调用了Camera对象的*getPicture*函数然后传入其中。这个函数讲会返回一个Promise，在执行完成之后，这个Promise将会给我们返回一个图片在设备上的

存储路径。现在我们只是用日志输出这个值，但是这个值后续会好好用上的。如果返回的是一个错误值的话，我们就用SimpleAlert来展示错误信息给用户了。
照片照好了，存放路径也返回了，临时文件夹里。这样图片临时显示出来了，照片不会长久的保存，因为临时目录会被随时清理掉。为解决此问题，我们需要把照片移动其他地方去，然后再次存储这个照片的新路径。我们可以用File插件来完成这个。

将照片移动到永久存储中

为了用上File插件，我们需要对*takePhoto*函数进行一些修改：

- 通过返回的 *imagePath* 来找到临时存储里面的照片
- 重命名，然后移动到设备上的'snapaday'文件夹（也就是永久存储）
- 基于这个新的位置新建一个照片对象

看来*takePhoto*函数会变得复杂了好多。因为要写这么多代码。我会对代码详细添加注释，当然也会一步步的去讲解。
> 修改 **takePhoto**里的 **getPicture** 调用如下：

```
Camera.getPicture(options).then((imagePath) => {
    //Grab the file name
    let currentName = imagePath.replace(/^.*[\\\/]/, '');
    //Create a new file name
    let d = new Date(),
    n = d.getTime(),
    newFileName = n + ".jpg";
    if(this.platform.is('ios')){
        //Move the file to permanent storage
        File.moveFile(cordova.file.tempDirectory, currentName,cordova.file.dataDirecto
ry, newFileName).then((success: any)=> {
            this.photoTaken = true;
            this.createPhoto(success.nativeURL);
            this.sharePhoto(success.nativeURL);
        }, (err) => {
            console.log(err);
            let alert = this.simpleAlert.createAlert('Oops!', 'Something went wrong.')
;
            alert.present();
        });
    } else {
        this.photoTaken = true;
        this.createPhoto(imagePath);
        this.sharePhoto(imagePath);
    }
},(err) => {
    let alert = this.simpleAlert.createAlert('Oops!', 'Something went wrong.');
    alert.present();
});
```

希望上面添加的注释可以让后面将要进行的东西变得简单些，这里有一个高级别的手把手的讲解，从camera返回*imagePath*后发生了什么：

1. 移除 imagePath 最后一个 / 前面的所有内容得到当前文件名
2. 使用日期新建一个唯一的文件名，这样我们不会覆盖任何东西
3. 如果我们是在iOS上运行的话，我们将照片从临时存储移动到持续存储中
4. 将 *photoTaken* 设为 *true*，将新路径传给图片然后传到 *createPhoto* 和 *sharePhoto*。

做完这些的话咱们的相片就存储到永久存储空间去了。记住，咱们现在还没有定义*createPhoto*和*sharePhoto*函数。我们现在来创建*createPhoto*函数，对于*sharePhoto*的话，我们留到整合社交分享插件的时候再来实现。
*createPhoto*函数接受一个路径参数（nativeURL本地路径）并在应用中持有他的引用。
> 修改 **createPhoto** 函数为如下：

```
createPhoto(photo): void {
    let newPhoto = new PhotoModel(photo, new Date());
    this.photos.unshift(newPhoto);
    this.save();
}
```

你也看到了好简单的说。我们在使用Photo Model新建相片实例的时候传入了一个相片的本地路径和创建日期，然后将新建的相片对象加入到*this.photos*数组。我们没有用**push**方法（因为是将相片添加到数组尾部），我们通过**unshift**将他添加到数组头部。我们这部做的原因是我们需要以反向的方式来展示图片（最新的最先显示），这么做我们的生活就更轻松。
同时我们也调用了*save*函数，他会将我们的数据保存到数据存储中，但是我们目前还没有实现这个函数。

更新模板

我们现在将相片添加到了*this.photos*数组，我们现在可以将模板更新为循环显示他们了。由于你可能是想通过浏览器测试，不能通过照相机来添加照片，那么请随意打开测试数据的注释，这样就可以看到本部分代码的效果。
我们现在来更新模板。
> 修改 **src/pages/home/home.html** 为如下：

```
<ion-header>
    <ion-navbar color="danger">
        <ion-title>
            <img src="assets/images/logo.png" />
        </ion-title>

        <ion-buttons end>
            <button ion-button icon-only (click)="playSlideshow()"><ion-icon name="pla
y"></ion-icon></button>
        </ion-buttons>
    </ion-navbar>
</ion-header>

<ion-content>
    <ion-list>
        <button ion-item *ngIf="!photoTaken" detail-none (click)="takePhoto()">
        <img src="assets/images/smile.png" />
        </button>

        <ion-item-sliding *ngFor="let photo of photos">
            <ion-item>
            <img [src]="photo.image" />
            <ion-badge item-right light>0 days ago</ion-badge>
            </ion-item>

            <ion-item-options>
            <button ion-button icon-only color="light" (click)="removePhoto(photo)"><i
on-icon name="trash"></ion-icon></button>
            </ion-item-options>
        </ion-item-sliding>
    </ion-list>
</ion-content>
```

对模板主要有两大变更。首先，我们用上了*ngFor*来村换相片数组，然后为每个项创建了一个入口。通过在*let photo of photos*中的*let photo*，我们可以得到当前渲染的照片的应用，即：photo.image访问照片的存储路径。

由于我们可以访问到每张照片的路径，我们就可以设置*img*元素来展示他了。注意我们用来方括号*[src]*，他允许我们将*photo.image*设置到image元素上来。这意味着会先评估*photo.image*，然后蛇之都奥**src**。如果我们不用方括号的话，**src**将是字符串"photo.image"。

我们也有一个删除按钮，他会将相片（也就是*let photo*）传入到*removePhoto*函数。我们现在就来实现这个方法吧。

> 修改 *src/pages/home/home.ts* 里的 *removePhoto* 函数：*

```
removePhoto(photo): void {
    let today = new Date();
    if(photo.date.setHours(0,0,0,0) === today.setHours(0,0,0,0)){
        this.photoTaken = false;
    }

    let index = this.photos.indexOf(photo);
    if(index > -1){
        this.photos.splice(index, 1);
        this.save();
    }
}
```

这个函数的第二部分够直白了，我们在photos数组中找到photo，移除他，触发save。第一部分看起来有点迷。这里搞事的是，如果用户拍勒个照，然后删除这个照片，他们就可以另外拍一张了（因为当日无照哇）。但是当他们删除了较早的照片的时候，他们就没法拍照。
所以啦，我们就获取被删除的照片的拍摄日期，然后跟今天日期进行对比。如果删除的是今日的照片，那么我们就把*this.photoTaken*设置为*false*这样用户今天有可以开心的拍另一张照片了。

总结

这节课实在是...！！！这节课设计了相当复杂的东西，但是我们在这节课中实现了应用的核心功能。还剩下一点点东西，但是我们现在可以拍照了，可以在将他们在一个列表中展示出来了，酷不酷！如果你想看看效果的话，到真机上去跑一下吧。如果你不知道我们做了些啥，可以直接跳到本书的 测试与调试部分去了解如何在设备上安装应用，然后回来完成应用剩余部分。
重要：如果现在想要在真机上测试拍照，那么你就需要将*loadPhotos*更新一下：

```
loadPhotos(): void {
    this.loaded = true;
}
```

你不能在数据加载完成之前拍照，所以我们得先伪造一下。在接下来的课程中，我们将学习如何创建一个数据服务来永久存储我们的照片，这样在用户后面回到应用的时候可以获取他们。

第五课：保存和加载照片

我们现在可以拍照，可以展示了，还可以删除了。问题是当应用关闭的时候，*this.photos*里面的数据会跟着丢失。显然，当用户下次使用应用的时候发现他们的自拍不见了，他们肯定是不高兴的，所以这节课就是来学习如何制作一个数据存储服务来持久存储数据和加载数据到应用中。
这个流程的大部分我们都设置好，我们已经生成了一个'Data'提供者，然后也已经在**home.ts**里面导入了，我们甚至还给他创建了一个save函数，这个函数是保存数据用的。所以，实际上我们要做的是这些：

- 实现save函数以调用数据服务
- 修改空空如也的Data，提供save和load功能
- 在应用打开的时候加载照片数据到应用中

实现数据服务

现在要给**data.ts**添加代码了让他可以保存数据到存储中。这个服务的代码看起来简单得难以置信，所以我们先瞄一下然后再讨论。
> 修改 **src/providers/data.ts** 为如下：

```
import { Storage } from '@ionic/storage';
import { Injectable } from '@angular/core';

@Injectable()
export class Data {
    constructor(public storage: Storage){

    }

    getData(): Promise<any> {
        return this.storage.get('photos');
    }

    save(data): void {
        let newData = JSON.stringify(data);
        this.storage.set('photos', newData);
    }
}
```

我们先看一下顶部的导入语句：

```
import { Storage } from '@ionic/storage';
```

Storage是Ionic的统一存储服务，他在提供统一的API的同时会选择最佳的存储方式。
在设备上运行的时候，如果可以用SQLite插件的时候（我们早先安装过），那么他就会使用本机SQLite数据库来存放数据。鉴于SQLite数据库只在设备本机上才能运行，在SQLite不可用的情况下，Storage还会用IndexedDB， WEBSQL或者标准的浏览器*localStorage*。
但是最好是用SQLite，因为基于浏览器的本地存储不是完全可靠的，存在着被操作系统擦除的风险。数据可能会随时被删掉显示是很难接受的。
我们看看**getData**函数：

```
getData(): Promise<any> {
    return this.storage.get('photos');
}
```

这个函数提供给我们获取存储的最新的数据。storage的get方法将返回一个promise，但是注意，我们没有在promise完成的时候做任何操作。这样我们可以在任意调用这个函数的地方处理就可以了，这样的话应用的工作流就更明了。（希望描述简单明了）
接着，我们有一个*save*函数，用来操作实际的数据存储：

```
save(data): void {
    let newData = JSON.stringify(data);
    this.storage.set('photos', newData);
}
```

我们将所有数据都存储成一个单独的JSON字符串，所以我们先调用了**JSON.stringify**函数然后通过storage的**set**方法存储。
这就是存储数据的全部，真心不复杂啊亲。现在我们来处理从存储中加载数据回应用。也就是我们接下来要处理的是*loadPhotos*函数。
> 修改 **src/pages/home/home.ts** 的 **loadPhotos** 函数：

```
loadPhotos(): void {
    this.dataService.getData().then((photos) => {
        let savedPhotos: any = false;

        if(typeof(photos) != "undefined"){
            savedPhotos = JSON.parse(photos);
        }

        if(savedPhotos){
            savedPhotos.forEach(savedPhoto => {
                this.photos.push(new PhotoModel(savedPhoto.image, new
                Date(savedPhoto.date)));
            });
        }

        if(this.photos.length > 0){
            let today = new Date();
            if(this.photos[0].date.setHours(0,0,0,0) === today.setHours(0,0,0,0)){
                this.photoTaken = true;
            }
        }
        this.loaded = true;
    });
}
```

我们首先调用了数据服务里的*getData()*函数。这个函数会以JSON字符串的方式返回所有照片数据，所以我们第一个要做到事情是对这个字符串进行JSON解码将他解析成一个可以直接使用的对象，如果没有照片数据返回的话应该是一个空数组。
如果存储中加载照片数据成功的话，我们就可以通过这些数据，给他们重建照片数据模型。
接着，我们又要判断用户今天是否还能拍照。我们检查第一位的照片的日期是否就是今天，如果是的话*photoTaken*就设为true。
一旦这些都执行完成，我们将*this.loaded*设为true来标记所有照片都加载完成。现在，我们只剩下*save*函数调用数据服务的函数。
我们移步来完成save函数的定义。
> 修改**home.ts**的**save**函数为如下：

```
save(): void {
    this.dataService.save(this.photos);
}
```

现在，当你调用*save*函数的时候，他都会把当前的*this.photos*传入到数据服务进行存储。

总结

这节课相当的简单，所以希望你能够从上一节课程中缓过来。保存和加载数据听起来很复杂，但是，你看，保存简单的数据还是蛮简单的。
下一节课，我们将做些稍微好玩的事情，我们将要完成Slideshow页，这样我们可以看到用户的所有照片的滑动展示页，同时完成的还有我们自定义的“Days Ago”管道。

第六课：新建一个自定义的管道和所有相片的飞页（Flipbook）

本节课中，我们将制作自定义管道用来更友好的展示照片的拍摄日期，我们也将完成应用的关键部分之一也就是slideshow。虽然他没有拍照那么复杂，但是他算是整个应用的灵魂。

创建一个自定义管道

我们在本书的基础部分已经涵盖了管道是什么，这里还是跟你简单温习一下：管道实际上允许我们在数据进行展示之前对他进行修改。
我们此处的目标是创建一个标签显示此照片上几天之前拍摄的，这样，“3 天前”，“10 天前”。目前而言，我们已经在照片的数据上存储了拍摄日期，不幸的是，他看起是酱紫的：

```
Sun Mar 06 2016 00:40:02 GMT+1000 (AEST)
```

也就是天底下最不友好的日期显示方式。所以，这里是**@Pipe**最好的使用情景：我们创建一个管道用来接收丑陋的日期格式，转换到“X 天前”这样的格式，然后返回之。
我们先看看如何创建这个管道，然后在慢慢讨论。
> 修改 **src/pipes/days-agp.ts** 为如下：

```
import { Injectable, Pipe } from '@angular/core';

@Pipe({
    name: 'daysAgo'
})
@Injectable()
export class DaysAgo {
    transform(value, args?) {
        let now = new Date();
        let oneDay = 24 * 60 * 60 * 1000;
        let diffDays = Math.round(Math.abs((value.getTime() - now.getTime())/(oneDay))
);

        return diffDays;
    }
}
```

@Pipe装饰器里面我们提佛那个一个名字*daysAgo*，意思就是在模板中可以直接通过使用这个名字作为关键字来使用这个管道。使用管道的时候，永远都需要传入数据，然后数据都会被传入*transform*函数也就是上面的*value*。我们会把我们的照片对象的Date对象传入到这个管

道，这就是我们这里要做的。同时注意可以通过管道传入额外的参数，也就是我们为什么用*args?*，因为这不是强制要求传入的参数。
首先我们来一点数学魔法得出照片上几天之前拍摄的，（Stack Overflow也有这个，我的解决方式可能比他丑点儿），然后返回他。返回的值就是将被渲染到模板上去的内容。
如果你还记得之前的课的话，我们已经在**app.module.ts**中设置好这个管道了。
> 修改**home.html**中的**photo**项为如下：

```
<ion-item>
<img [src]="photo.image" />
<ion-badge item-right light>{{photo.date | daysAgo}} days ago</ion-badge>
</ion-item>
```

注意了（译者敲了下黑板），我们加了这个句：

```
{{photo.date | daysAgo}}
```

这样写会把*photo.date*传入到*daysAgo*管道，然后将*daysAgo*返回的东西展示最这里。最终结果是一个带有类似“5 天前”的徽章。

为所有照片创建一个**Slideshow**

我们已经创建好了Slideshow的模板，所以现在我们需要制作一些逻辑来循环和展示所有照片，同时我们也要添加一个打开slideshow的途径（重启途径也要）。
我们还是先从*playSlideshow*函数开始，这样我们可以真正的打开页面。
> 修改 **src/pages/home/home.ts**的 **playSlideshow** 函数为如下：

```
playSlideshow(): void {
    if(this.photos.length > 1){
        let modal = this.modalCtrl.create(SlideshowPage, {photos:
        this.photos});
        modal.present();
    } else {
        let alert = this.simpleAlert.createAlert('Oops!', 'You need at least two photo
s before you can play a slideshow.');
        alert.present();
    }
}
```

跟创建警告提示框类似，我们给用户撞见了一个模态框页面。首先，我们创建了一个Modal，然后通过NavController呈现到用户面前。在这里，我们使用已经创建好的SlideshowPage来创建模态框，也将所有的照片数据传入进去。这样我们可以在Slideshow页面上获取到这些数据。

注意，我们只有在用户有超过1张照片的时候才会触发他（因为没有或者只有1张照片的slideshow就不是真的slideshow了，对吧？），否则会显示警告提示框。
现在，我们来定义Slideshow页的类定义。这个类挺小的，所以先全部贴出来，然后分段讲解。

> 修改 **src/pages/slideshow/slideshow.ts** 为如下：

```
import { NavParams, ViewController } from 'ionic-angular';
import { Component, ElementRef, ViewChild } from '@angular/core';

@Component({
    selector: 'page-slideshow',
    templateUrl: 'slideshow.html'
})
export class SlideshowPage {
    @ViewChild('imagePlayer') imagePlayer: ElementRef;
    imagePlayerInterval: any;
    photos: any;

    constructor(public navParams: NavParams, public viewCtrl: ViewController) {
        this.photos = this.navParams.get('photos');
    }

    ionViewDidEnter(){
        this.playPhotos();
    }

    closeModal(){
        this.viewCtrl.dismiss();
    }

    playPhotos(){
        let imagePlayer = this.imagePlayer.nativeElement;
        let i = 0;
        //Clear any interval already set
        clearInterval(this.imagePlayerInterval);
        //Restart
        this.imagePlayerInterval = setInterval(() => {
            if(i < this.photos.length){
                imagePlayer.src = this.photos[i].image;
                i++;
            }
            else {
                clearInterval(this.imagePlayerInterval);
            }
        }, 500);
    }
}
```

这里有些东西可能你不大熟悉。我们导入了NavParams服务可以用他拿到在创建的时候传过来的数据。你可以看到我们在构造器里面通过*this.navParams.get('photos');*获取数据。这里另一个奇怪的东西是**ViewController**。我们需要用到这个来隐藏模态框（我们这里定义了一个*closeModal*函数提供给模板里面的按钮使用）。
这里最重要的函数是*playPhotos*，这就是我们循环所有照片然后在页面上相应改变对于的图片元素的地方。首先，我们通过之前设置好的*@ViewChild*引用得到图片元素（之前模板里面通过 #本地变量 的方式定义的）的引用，我们清理了当前正在运行的所有定时器（防止用户在slideshow还在播放的时候突然重启slideshow）。然后我们开始循环*this.photos*里的照片。如果还剩下照片可以展示，图片的**src**属性将会使用下一张照片的数据更新，如果没有的话，就会清理计时器。上面代码中，定时器的间隔是500毫秒，也就是0.5秒，我们可以根据个人喜好通过调整这个数值来调整slideshow的快和慢。（甚至给用户提供选择来控制）
*playPhotos*函数上*ionViewDidEnter*函数自动触发的，*ionViewDisEnter*函数上在进入视图的时候自动执行的，这个函数在我们每次进入视图的时候都会执行。

总结

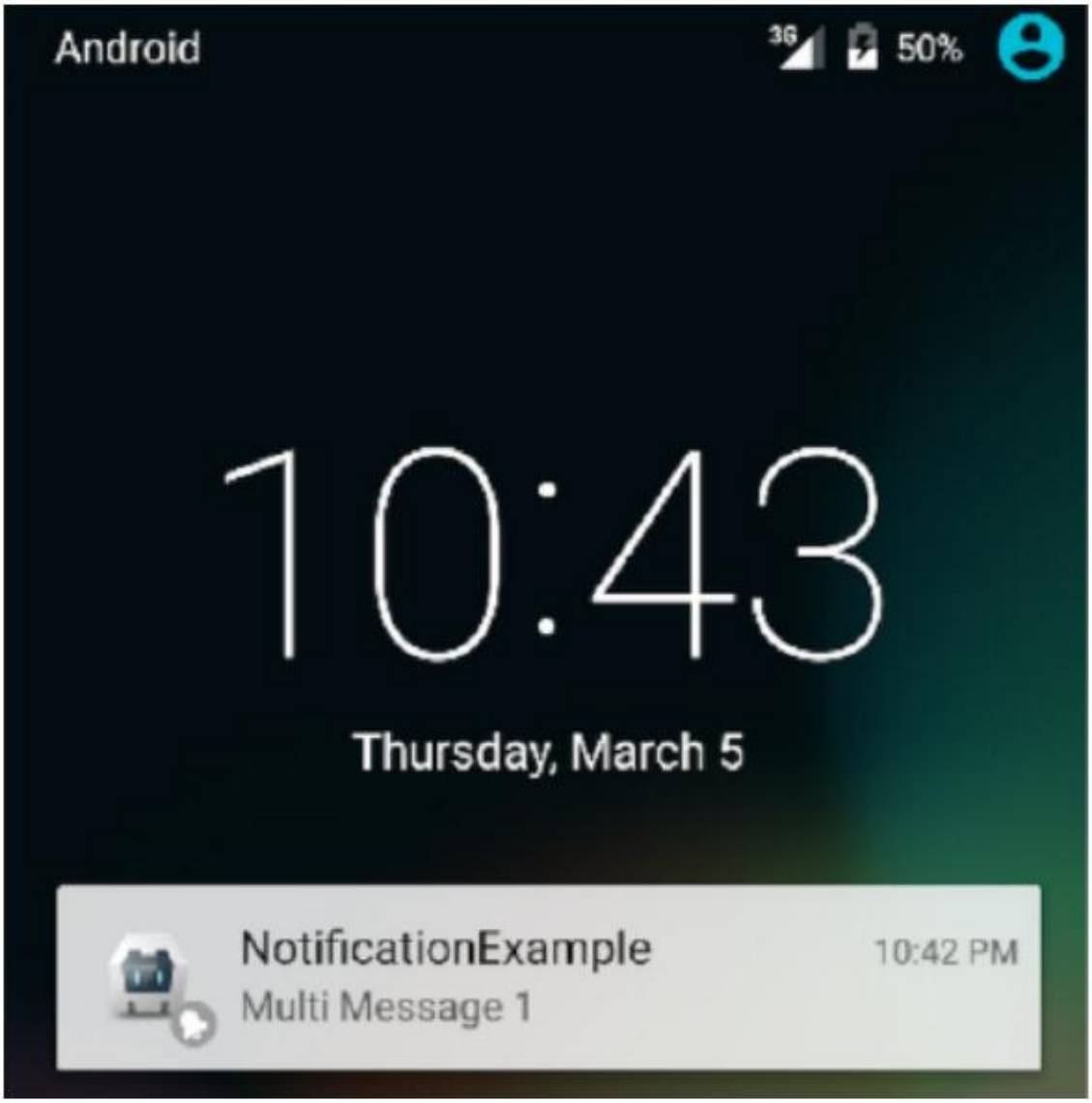
这是另一节比较小的课，但是这里讲的东西都蛮酷的。至此，应用的大部分主体功能都完成了，我们只要加上一些边边角角就可以了。当然，我们还是得梅花他，但是下节课我们要学习的是整合本地通知和社交分享功能。

第七课：整合本地通知与社交分享

目前为止，我们完成了应用的主体功能的制作：我们可以拍照，将他们展示到一个列表上，删除照片，从存储加载照片，播放一个slideshow等等。我们现在要加入的功能是“有了更好”的这种类型的功能，可以用来改善我们的用户体验。
我们将添加本地通知功能，用于向提醒用户记得每天拍个照，我们也会带来社交分享功能这样用户可以跟他们的朋友分享他们的照片。在准备工作部分我们已经安装好了这些插件，所以现在我们只要实现功能就可以了。
预备，开始！

本地通知Local Notification

在实现本地通知之前，给大家普及一下推送通知**push notification**和本地通知**local notification**的区别。他们看起来的行为是一致的，他允许你向用户提醒是否打开了应用，大概这样子：



不同的是，推送通知需要一个外部服务器控制向设备发送通知，但是本地通知完全是设备本身去控制的。这意味着，对于闹钟，排程通知或者类似咱们应用的每日提醒来讲，本地通知是再合适不过了。推送通知更适合用来提醒设备外面发生的事情，例如有人在Facebook给你发了消息之类的。

我们只需要在根组件里面添加一点点代码就可以实现这个功能。

> 修改 **src/app/app.component.ts**为如下：

```
import { Component } from '@angular/core';
import { Platform } from 'ionic-angular';
import { HomePage } from '../pages/home/home';
import { LocalNotifications } from 'ionic-native';

@Component({
    template: `<ion-nav [root]="rootPage"></ion-nav>`
})
export class MyApp {
    rootPage = HomePage;

    constructor(platform: Platform) {
        platform.ready().then(() => {
            if(platform.is('cordova')){
                LocalNotifications.isScheduled(1).then( (scheduled) => {
                    if(!scheduled){
                        let firstNotificationTime = new Date();
                        firstNotificationTime.setHours(firstNotificationTime.getHours(
)+24);

                        LocalNotifications.schedule({
                            id: 1,
                            title: 'Snapaday',
                            text: 'Have you taken your snap today?',
                            at: firstNotificationTime,
                            every: 'day'
                        });
                    }
                });
            }
        });
    }
}
```

由于这个构造器是打开应用第一个运行的，我们把插件的调用发哦到了*platform.ready()*里面以确保插件准备可用。然后我们检查了当前运行平台是否是'cordova'，就跟之前的Camera插件一样，因为这个插件只能在眞机上运行。

然后，我们就来到了使用插件的地方，我们可以访问到来自Ionic Native的*LocalNotification*。我们先检查一下id为1的排程是否被占用，我们已被占用，我们就啥都不干了，如果没有的话，我们通过*schedule*方法创建一个新的通知。

我们简单的提供了一个标题和一个需要展示的信息，同时也提供了一个id这样可以在后面用来区分这个通知（就像刚刚检查是否有排程一样）。我们也需要描述啥时候展示这个通知，所以我们指定了从现在起的24小时后（也就是明天这个时候会提醒），将频率设为每日，这样他会持续每天提醒（还有其他可用选项，例如每周）。

这是使用此插件的全部了，最起码我们要做的，如果想要实现更复杂的功能，那么请参考插件文档，查看更多可用选项。

社交分享

我们现在来添加社交分享插件。这是个非常好的插件，用于通过网络向不同的社交媒体分享数据，甚至SMS和邮件。手动整合不同的API（例如Facebook，Twitter等等）很烦的说，这个插件很好的解决了这些问题。
你可以分享一些东西到指定平台，或者你可以触发一个统一分享弹出框供用户选择分享平台。为实现此功能，我们现在要完善*takePhoto*函数已经调用了的*sharePhoto*函数。
> 修改 **src/pages/home/home.ts** 的 **Ionic Native**导入部分为如下：

```
import { Camera, File, SocialSharing } from 'ionic-native';
```

> 修改 **src/pages/home/home.ts** 的 **sharePhoto**函数为如下：

```
sharePhoto(image): void {
    let alert = this.alertCtrl.create({
        title: 'Nice one!',
        message: 'You\'ve taken your photo for today, would you also like to share it?'
,
        buttons: [
            {
                text: 'No, Thanks'
            },
            {
                text: 'Share',
                handler: () => {
                    SocialSharing.share('I\'m taking a selfie every day with #Snapaday'
, null, image, null)
                }
            }
        ]
    });

    alert.present();
}
```

我们之前也创建过一些警告框，那些警告框都是通过Simple Alert服务来创建的。我们这里不用Simple Alert来创建的原因是我们现在要创建的警告框复杂多了。这里的警告框有两个按钮，其中一个按钮会通过社交分享插件车发分享（另一个只是关闭这个框）。
你也看到了‘Share’按钮的处理器，我们也只是调用了插件（通过*window.plugins.socialsharing*）的*share*方法传入一条信息和图片。这个插件有大量不同的配置可用，所以，先读读文档吧，勇敢的少年哟。
现在当用户点击‘Share’的时候将会弹出一个框，上面显示了要分享的图片，然后你可以在弹出框上选择需要分享到的平台。

总结

我们添加的功能都有助于应用吸引用户。即使用户每天想要来个自拍，没有提醒功能的话，忘掉的几率还是比较大的。轻松分享照片到社交媒体的能力可以帮助将应用推广到全世界希望能够带来更多的下载量。
接下来只剩下样式定义这一个事情要做了，他可以让我们的应用看起来没那么丑。

第八课：自定义样式

别急，咱们快做完了，实际上应用基本上已经完成了，这节课之后所有功能将会被完善，应用看起来也会漂亮多了。
如果你记得的话，在基础部分我们学习过添加样式的几种方式。如果你跳过了那个部分，或者对我讲的不是很了解，建议你回头去看一遍基础部分的主题定制部分。
> 修改 **theme/variables.scss** 里的命名颜色变量：

```
$colors: (
    primary: #387ef5,
    secondary: #32db64,
    danger: #fa2d18,
    light: #f4f4f4,
    dark: #222,
    favorite: #69BB7B
);

$item-ios-padding-left: 0;
$item-md-padding-left: 0;
$item-ios-padding-right: 0;
$item-md-padding-right: 0;
$item-ios-padding-top: 0;
$item-md-padding-top: 0;
$item-ios-padding-bottom: 0;
$item-md-padding-bottom: 0;
$list-ios-margin-bottom: 0px;

$list-ios-margin-top: 0px;
$list-md-margin-bottom: 0px;
$list-md-margin-top: 0px;
```

首先，我们重新定义了应用模板中会用到的颜色。记住，这些颜色可以这样在模板中使用：
然后我们覆盖了大量默认的SASS变量，这些变量基本就是移除列表和列表项的间隔和边距这样照片就可以填满整个可用区间了。
继续前进来到了组件指定样式。我们将稍稍的改动模板来纳入一些自定义的类，并在每个组件对应的*.scss*中定义好这些类。我们先从主页开始。
> 修改 **src/pages/home/home.html** 如下：

```
<ion-header>
    <ion-navbar color="danger">
        <ion-title>
            <img src="assets/images/logo.png" />
        </ion-title>

        <ion-buttons end>
            <button ion-button icon-only (click)="playSlideshow()"><ion-icon name="pla
y"></ion-icon></button>
        </ion-buttons>
    </ion-navbar>
</ion-header>

<ion-content>
<ion-list>
    <button ion-item *ngIf="!photoTaken" detail-none (click)="takePhoto()">
        <img src="assets/images/smile.png" class="list-photo" />
    </button>

    <ion-item-sliding *ngFor="let photo of photos">
        <ion-item>
            <img [src]="photo.image" class="list-photo" />
            <ion-badge item-right color="light">{{photo.date | daysAgo}} days ago</ion
-badge>
        </ion-item>
        <ion-item-options>
            <button ion-button icon-only color="light" (click)="removePhoto(photo)"><i
on-icon
            name="trash"></ion-icon></button>
        </ion-item-options>
    </ion-item-sliding>
</ion-list>
</ion-content>
```

> 修改 **src/pages/home/home.scss** 如下：

```
page-home {
    .scroll-content {
        background-color: #222222;
    }
    .logo {
     max-height: 39px;
    }
    .list-photo {
        width: 100%;
        height: auto;
    }
    ion-badge {
        position: absolute;
        right: 10px;
        top: 5px;
    }
}
```

这里没啥值得激动的事情。我们确保了照片填满整个宽度，给content区域和列表项设置了一个黑色的背景色，使用绝对定位让我们的“days ago”显示到对应的地方。
现在来看看slideshow页面。
> 修改 **src/pages/slideshow/slideshow.scss** 为如下：

```
page-slideshow {

    .scroll-content {
        background-color: #222222;
    }

    .image-container {
        position: absolute;
        top: 0;
        bottom: 0;
        left: 0;
        right: 0;
        display: flex;
        justify-content: center;
        align-items: center;
    }
    #imagePlayer {
        width: 100%;
        height: auto;
        vertical-align: middle;
    }
}
```

这里的样式只是让照片显示在水平和垂直方向上都居中（这不是看起来的那么简单！）。我们用flexbox耍了点小花招才实现。Flexbox稍微有点高级，如果你想看看他的入门介绍的话，请看这个。

如果现在看一眼应用的话，现在应该是这样的：

你也看到了，我们并没有加很多的样式，但是整个应用看起来好看多了。

结论

恭喜完成了Snapaday的学习和制作。开发这个应用的过程中，我们学到了很多知识，主要是以下这些：

- 如何整合本地插件
- 如何使用Camera API
- 如何使用File API
- 如何使用本地通知
- 如何使用模态框
- 如何创建自定义的provider
- 如何永久存储数据

改进的空间永远都存在，特别是当你学习事物的时候。遵循指导手册固然很好，但是自己去学习弄清楚一些事情就更完美了。希望你有足够的背景知识来自己完成一些功能扩展，以下是一些想法：

- 为应用创建自己的自定义主题【简单】
- 修改‘Days Ago’管道以展示‘taoday’和‘1 day ago’而不是‘0 days ago’和‘1 days ago’【中等】
- 添加设置允许用户控制slideshow的播放速度【困难】
- 允许用户控制是否需要每天通知和他们什么时候需要通知【困难】

记住，在你学习的过程中，Ionic 2文档是你最好的朋友。

接下来？

我们已经有一个完成的应用了，但是这不是故事的结局。你需要让他运行到真实设备上去，提交到应用商店，这不是个简单的任务。本书的最后部分会带你完成这些内容。你可以先看看。

项目：露营伴侣（Camper Mate）

第一课：介绍

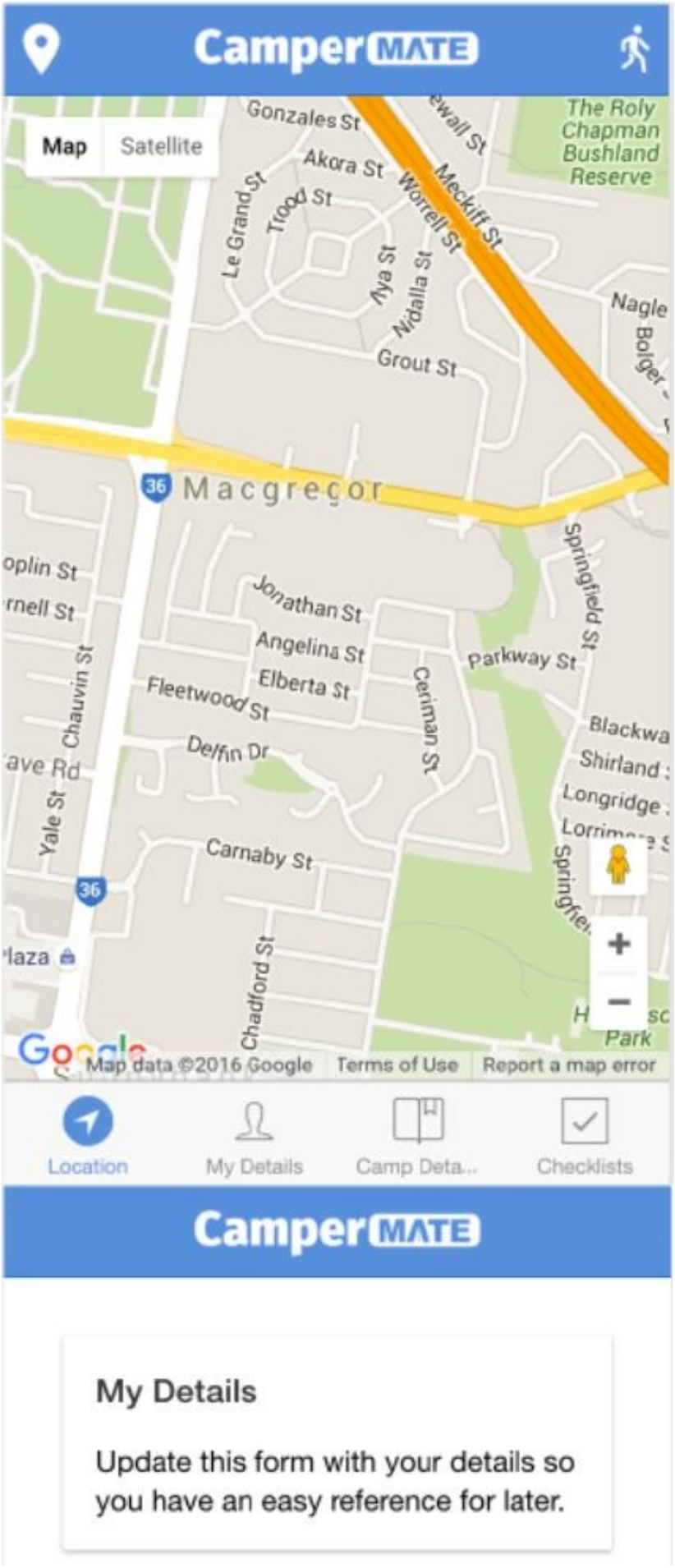
Camper Mate是一个制作起来非常有趣的应用，因为他跟其他应用不一样，他没有什么特定的目的，他应该只是一个工具带类的app。他给大篷车或者野外露营用户提供了很多有趣的功能。
这就给了我们一个机会来做一些其他应用没做过的事情，同时他也提供了一些挑战 -- 其中一个就是我们有很多不同的数据需要去存储，而不是一套数据，所以我们的数据模型比之前创建的复杂得多。
最重要的两个只是点数整合使用Google Maps和通过表单获取用户输入。同时，本书的第一个应用Quicklist可以非常好的作为这个应用的一个补充，所以我们也会学习怎么样把整个Quick Lists应用天建到Camper Mate的一个标签页中（无法想象的简单）。
本应用的具体功能包括：

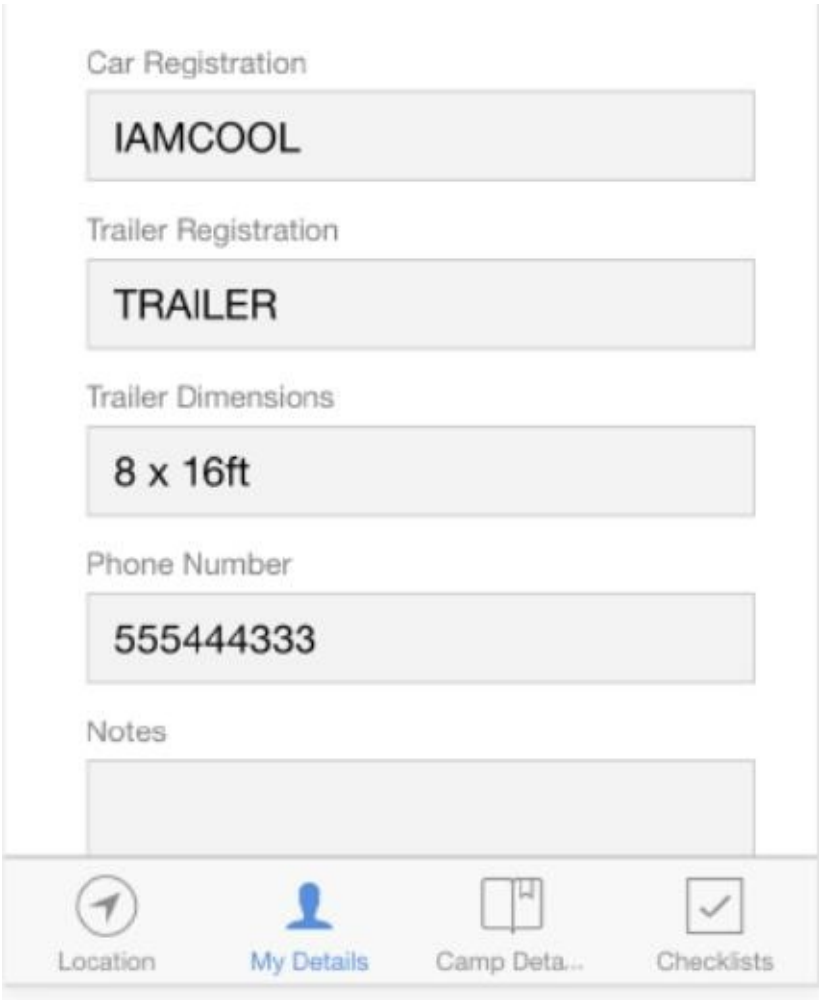
- 一个给用户设置露营地点的地图。当他们离开营地的时候可以点击一个按钮来展示如何返回营地。
- 一个给用户存储个人露营相关细节的表单（车和拖车注册信息）
- 一个给用户存储个人露营地点相关信息的表单（访问代码，WiFi密码）
- 创建，修改和删除自定义checklist的能力（这部分功能Quick List提供）

以及一些会涵盖的概念：

- 创建表单获取输入
- 实现Google Maps创建provider对他进行操作
- 创建一个标签页布局
- 保存和获取不同套的数据
- 重用其他应用的组件

以下是一些最终效果图：


CamperMATE
Map
Satellite
Gonzales St
Akora St
Trood St
Le Grand St
Worrell St
Meckiff St
The Roly Chapman Bushland Reserve
Aya St
Nidalla St
Grout St
Nagle
36
Macgregor
Jonathan St
Angelina St
Elberta St
Fleetwood St
Delfin Dr
Parkway St
Springfield St
Cerman St
Chauvin St
Yale St
Carnaby St
Chadford St
Blackwa
Shirland
Longridge
Park
Map data ©2016 Google
Terms of Use
Report a map error
Location
My Details
Camp Deta...
Checklists
CamperMATE
My Details
Update this form with your details so you have an easy reference for later.



课程结构

1. 准备工作
2. 创建一个标签页布局
3. 用户输入域表单
4. 实现Google Maps和Geolocation
5. 保存和获取数据
6. 重用组件
7. 自定义样式

准备好了吗？
现在你知道了你要做什么，那么我们就可以开始了。

第二课：准备工作

本课是在旅程继续进行之前的一些准备工作。我们要生成应用，设置所有组件和需要用到的Cordova插件。完成本课之后我们应该有一个万事俱备的项目骨架，可以接着进行编码工作。开始新项目的第一准则是确保使用的是最新版的Ionic和Cordova，如果最近没有更新过的话可以运行如下命令：

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

或者

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

如果在安装Ionic或者生产新项目的时候遇到任何问题的话，检查下是否安装了最新版的Node。安装完之后，再次安装ionic之前请运行：

```
npm uninstall -g ionic npm cache clean
```

生成新应用

本应用将使用空白初始模板，跟名字说的一样，就是个空的Ionic项目。但是会有一个内置的页面名为**home**，我们下节课中将用作列表显示页。
> 运行如下命令生成新项目：

```
ionic start campermate blank --v2
```

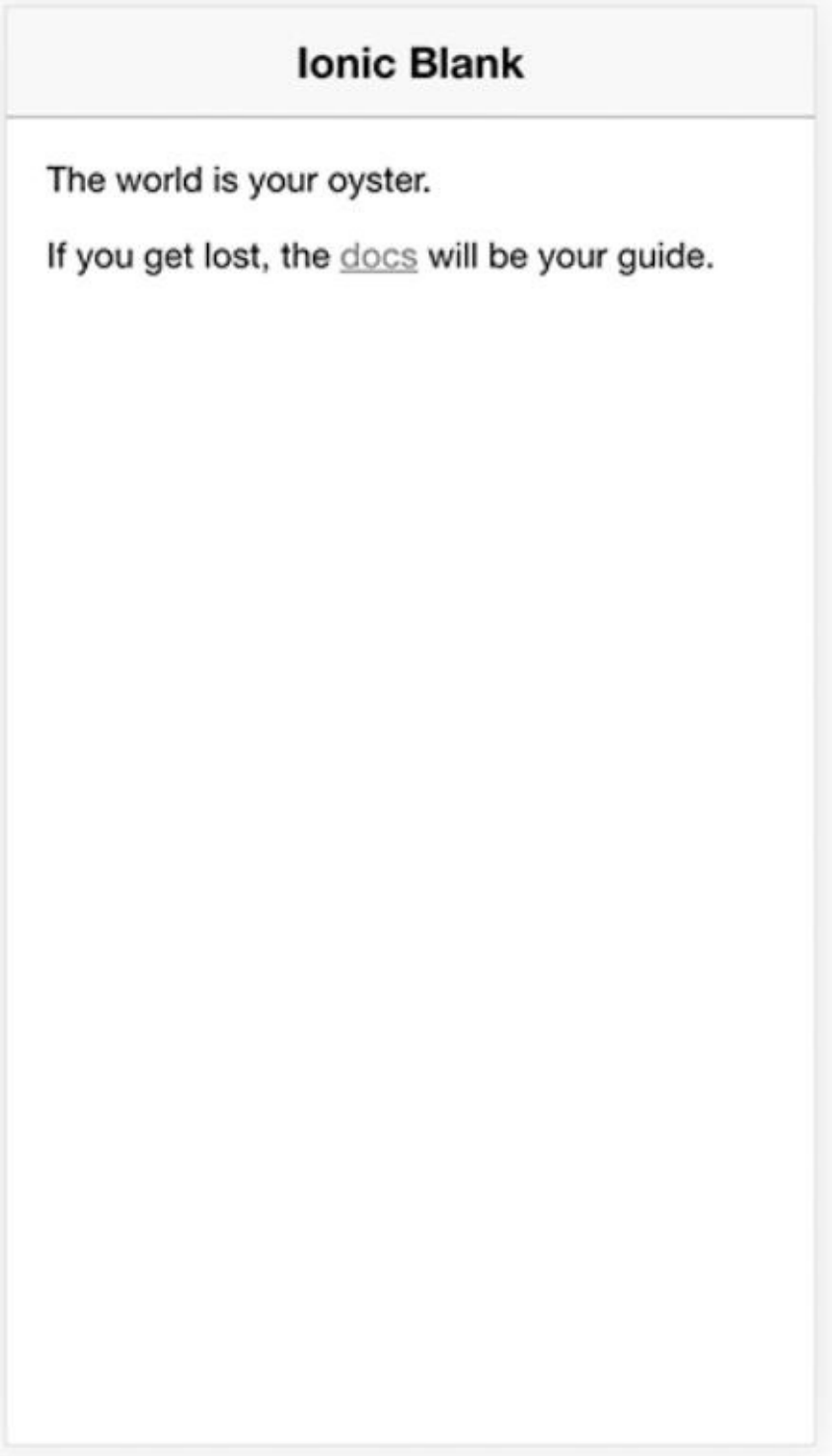
> 运行如下命令将新生成的项目作为当前目录：

```
cd campermate
```

现在可以在你中意的编辑器中打开这个项目了。通过以下命令可以预览创建的应用：

```
ionic serve
```

看起来是这样的：



创建需要的组件

我们来创建本应用的一些页面，我们需要重用到自动生成的home页面来创建我们的标签页布局（允许页面之间来回切换），但是我们还是需要添加一些其他的页面：location，个人细节以及营地细节三个标签页。 **>** 运行如下命令生成 **Location** 页面：

```
ionic g page Location
```

> 运行如下命令生成 **My Details** 页面：

```
ionic g page MyDetails
```

> 运行如下命令生成 **Camp Details** 页面：

```
ionic g page CampDetails
```

创建需求的服务

跟标签页一样，我们同时也需要去创建一些服务。我们将创建一个数据服务来保存和获取数据，一个Google Map服务用来负责整合Google Maps，一个Connectivity服务用来检查用户是否在线。

> 运行如下命令生成 **Data** 提供者：

```
ionic g provider Data
```

> 运行如下命令生成 **Google Maps** 提供者：

```
ionic g provider GoogleMaps
```

> 运行如下命令生成 **Connectivity** 提供者：

```
ionic g provider Connectivity
```

往App Module里面添加页面与服务

为了能够在项目里面可以使用这些页面和服务，我们需要将它们添加到**app.module.ts**文件里。所有我们自己创建的页面都需要添加到*declarations*数组和*entryComponents*数组里，所有我们创建的数据提供者都需要添加到*providers*数组，其他自定义组件或者管道（pipe）只需要添加到*declarations*数组即可。我们的数据模型只是一个简单的类，我们需要在任何地方使用，所以不用在模组里面设置。

> 修改**src/app/app.module.ts**到以下：

```
import { NgModule } from '@angular/core';
import { IonicApp, IonicModule } from 'ionic-angular';
import { MyApp } from './app.component';
import { Storage } from '@ionic/storage';
import { HomePage } from '../pages/home/home';
import { LocationPage } from '../pages/location/location';
import { MyDetailsPage } from '../pages/my-details/my-details';
import { CampDetailsPage } from '../pages/camp-details/camp-details';
import { GoogleMaps } from '../providers/google-maps';
import { Connectivity } from '../providers/connectivity';
import { Data } from '../providers/data';

@NgModule({
    declarations: [
        MyApp,
        HomePage,
        LocationPage,
        MyDetailsPage,
        CampDetailsPage
    ],
    imports: [
        IonicModule.forRoot(MyApp)
    ],
    bootstrap: [IonicApp],
    entryComponents: [
        MyApp,
        HomePage,
        LocationPage,
        MyDetailsPage,
        CampDetailsPage
    ],
    providers: [Storage, Data, GoogleMaps, Connectivity]
})
export class AppModule {}
```

注意，我们除了自己创建的*Data*提供者之外，我们还加入了一个*Storage*。Storage是Ionic提供的，可以通过它保存和获取数据 -- 我们后续会用到。

添加需要的平台

在给指定平台制作应用之前，你需要将它们添加到你的项目。
> 运行以下命令添加**iOS**平台：

```
ionic platform add ios
```

> 运行以下命令添加**Android**平台：

```
ionic platform add android
```

添加需要的Cordova插件

这个应用将会用到一些不同的Cordova插件。记住，Cordova插件只能在真机上使用。我会在添加他们的时候解释他们的用途。
> 运行以下命令添加**Geolocation**插件：

```
ionic plugin add cordova-plugin-geolocation
```

Geolocation插件允许我们获取用户当前位置，同时他也提供了长时间追踪用户位置的能力。

> 运行以下命令添加**Network**插件：

```
ionic plugin add cordova-plugin-network-information
```

这个插件可以让我们知道用户的当前网络信息，例如当前的连接类型。这让我们可以更精确的指导用户当前是否有可以激活的互联网连接。
> 运行以下命令添加**SQLite**插件：

```
ionic plugin add cordova-sqlite-storage
```

这个插件让你可以访问本地存储SQLite数据库。我们在此应用中添加他的原因是Ionic本地存储服务可以使用插件提供的稳定输出存储。

> 运行以下命令添加**App Browser**插件：

```
ionic plugin add cordova-plugin-inappbrowser
```

这个插件让我们可以通过他提供的webview来从外部网站。
> 运行以下命令添加**Status Bar**插件：

```
ionic plugin add cordova-plugin-statusbar
```

我们给所有项目添加此插件用来在应用中控制状态栏（设备屏幕顶部的状态条，包括时间，电池信息等等）。
> 运行以下命令添加**Splash Screen**插件：

```
ionic plugin add cordova-plugin-splashscreen
```

此插件允许我们控制闪屏（打开应用的时候的全屏画面）。

> 运行以下命令添加**Whitelist**插件：

```
ionic plugin add cordova-plugin-whitelist
```

所有应用会用到这个插件，他定义了应用里可以加载什么样的资源。没有他的话，你尝试加载的资源都会不成功。
添加了这个插件后，你也需要到**index.html**中定一个一个“Content Security Policy”。我们将添加一个非常宽松的策略实际上允许我们加载任何资源。基于你的应用，你可以提供一个更严格的策略，但是对于开发而言开放性策略就可以了。
> 修改 **src/index.html**文件，添加一下**meta**标签：

```
<meta http-equiv="Content-Security-Policy" content="font-src 'self' data:;
img-src * data:; default-src gap://ready file://* *; script-src 'self'
'unsafe-inline' 'unsafe-eval' * ; style-src 'self' 'unsafe-inline' *">
```

> 运行以下命令添加**Crosswalk**插件：

```
ionic plugin add cordova-plugin-crosswalk-webview
```

这个另一个每个应用都要添加的插件，但是你也可以先不添加。添加了这个插件后，在你编译Android的时候将会使用“Crosswalk”。Android有很多问题，特别是老设备，因为有太多不同的团建版本，不同的版本有不同的浏览器（记住，鉴于我们是制作HTML5应用，他实际上就是一个搭载的浏览器用来运行我们的应用）。Crosswalk做的是将一个现代的浏览器打包到应用中，这样一来应用无论是运行在什么设备上，都会使用相同的浏览器来运行，并且Crosswalk浏览器可以很好改善执行效率。

唯一的不足之处就是你的应用尺寸明显的变大了很多。总体上，我觉得这很值得，我也建议你使用他，如果你接受不了的话，也可以不用。更多关于Crosswalk Project的信息，请参考网站：https://crosswalk-project.org/

设置图片

制作此应用的时候，会用到一些图片。你下载的包里面已经包含了这些图片，但是你需要去生成的项目里面设置好他们。

> 将下载包 **src/assets**文件夹下面的**images**文件夹复制到应用里的 **src/assets**下面

总结

就这样！我们设置好了，准备好继续前几，现在我们开始进入到有趣的部分了。

第三课：新建一个标签页布局

本课中我们给应用创建一个基本的布局，也就是一个标签页布局。标签页的工作方式跟之前的页面有点不一样，页面只是标签页的持有者，但是实际内容是你导入进来的其他标签页面。实际工作中看到的话你就会知道了。
首先，我们得创建三个子标签来持有应用的不用页，但是实际上我们有一个额外的标签也是用于持有Quick List功能的。
我们先从持有所有标签页的Home页开始。
>修改 **src/pages/home/home.html** 为如下：

```
<ion-tabs>
    <ion-tab [root]="tab1Root" tabTitle="Location" tabIcon="navigate"></ion-tab>
    <ion-tab [root]="tab2Root" tabTitle="My Details" tabIcon="person"></ion-tab>
    <ion-tab [root]="tab3Root" tabTitle="Camp Details" tabIcon="bookmarks"></ion-tab>
</ion-tabs>
```

如你所见，此处布局非常简单。我们用到了接着通过设置了一些标签页。我们然后将这些标签页的*root*属性设置了一个表达式，这些表达式马上会在**home.ts**上定义。这个表达式将是标签页用来容纳的内容。我们也添加了title和icon用来展示在标签界面上。
现在，我们来看看类定义。
>修改 **src/pages/home/home.ts** 为如下：

```
import { Component } from '@angular/core';
import { LocationPage } from '../location/location';
import { MyDetailsPage } from '../my-details/my-details';
import { CampDetailsPage } from '../camp-details/camp-details';

@Component({
    selector: 'page-home',
    templateUrl: 'home.html'
})
export class HomePage {
    tab1Root: any = LocationPage;
    tab2Root: any = MyDetailsPage;
    tab3Root: any = CampDetailsPage;

    constructor(){

    }

}
```

我们导入了所有要在标签页中展示的页面，然后设置好了模板中要用到的表达式。第一个展示标签页展示的是**LocationPage**。
我们的标签页基本上是设置好了（万岁！），但是还没到休息时间 -- 我们需要给每个标签页设置好类。我们先从Location页开到。Location页面有一个地图，一些按钮给用户设置他们位置，给用户启动回家导航。
>修改 **src/pages/location/location.html** 为如下：

```
<ion-header>
    <ion-navbar color="primary">
        <ion-title>
            <img src = "assets/images/logo.png" class="logo" />
        </ion-title>
        <ion-buttons start>
            <button ion-button icon-only (click)="setLocation()"><ion-icon name="pin">
</ion-icon></button>
        </ion-buttons>
        <ion-buttons end>
            <button ion-button icon-only (click)="takeMeHome()"><ion-icon name="walk">
</ion-icon></button>
        </ion-buttons>
    </ion-navbar>
</ion-header>

<ion-content>

    <div #pleaseConnect id="please-connect">
        <p>Please connect to the Internet...</p>
    </div>

    <div #map id="map"></div>
</ion-content>
```

允许我们在应用顶部添加一个页首用于持有按钮，标题甚至在需要的时候直接整合Ionic导航系统的后退按钮。我们给他添加了*primary*属性用我们的*primary*颜色对他定制样式，这个颜色在**app.variable.scss**定义好了（我们后面会配置）。
在这个navbar里面我们用到了，他就是用来展示当前页标题和logo的。我们给他提供了*start*属性，这样按钮都会从左边开始排列，如果是*right*属性的话就是从右边开始排列。我们在两边各放置了一个按钮用处触发“设置位置”和“带我回家”功能。
最后，内容区域只有一个*div*我们在其中添加了*#map* -- 这让我们后面可以在类中获取到这个节点。同时*div*稍后也会作为我们Google Map的容器。我们添加了另一个*div*用来展示用户的连接信息告诉用户是否联网了（稍后实现这个）。
现在，我们来看看location页的类定义。
>修改 **src/pages/location/location.ts** 为如下：

```
import { NavController, Platform, AlertController } from 'ionic-angular';
import { Component, ElementRef, ViewChild } from '@angular/core';
import { Geolocation } from 'ionic-native';
import { GoogleMaps } from '../../providers/google-maps';
import { Data } from '../../providers/data';

@Component({
    selector: 'page-location',
    templateUrl: 'location.html'
})
export class LocationPage {

    @ViewChild('map') mapElement: ElementRef;
    @ViewChild('pleaseConnect') pleaseConnect: ElementRef;

    latitude: number;
    longitude: number;
    constructor(public navCtrl: NavController, public maps: GoogleMaps, public platfor
m: Platform, public dataService: Data, public alertCtrl:AlertController) {
    }

    ionViewDidLoad(): void {
    }

    setLocation(): void {
    }

    takeMeHome(): void {
    }
}
```

虽然我们设置了一些东西，但是还不足以继续。我们从Ionic Native中导入了稍后会用到的Geolocation，同样也导入了Google Maps和Data服务（稍后创建）。
我们把所有这些服务都注入到构造器中通过添加*public*将他们设置为成员变量，我们也定义了两个变量*latitude*和*longitude*，将用于持有用户位置。最后，我们添加了两个模板中按钮要调用的函数，目前还没有实现。
这里最奇怪的是使用了**@ViewChild**。你可以用他来获取模板中的元素的引用。所以，如果我们这样：

```
@ViewChild('map') mapElement: ElementRef;
```

他将会去**location.html**模板中查找一个含有*#map*的元素。如果找到了的话，将会返回一个*ElementRef*也就是那个元素的引用。在这个实例中，我们将这个引用指向到了*mapElement*。稍后，我们会把这个引用传递给Google Maps提供者。重点记住，这一步需要在*ionViewDidLoad()*方法里面做，因为这个函数只会在DOM（页面元素）完全加载完成的时

候运行一次。

现在，我们移步到Camp Details页。

> 修改 **src/pages/camp-details/camp-details.html** 页面为如下：

```
<ion-header>
    <ion-navbar color="primary">
        <ion-title>
            <img src="assets/images/logo.png" class="logo" />
        </ion-title>
    </ion-navbar>
</ion-header>

<ion-content padding>
    <ion-card>
        <ion-card-header>
            Camp Details
        </ion-card-header>

        <ion-card-content>
        Update this form with the details of your current camp so you have an
        easy reference for later.
        </ion-card-content>
    </ion-card>
    <ion-list no-lines>

        <ion-item>
            <ion-label stacked>Gate Access Code</ion-label>
            <ion-input type="text"></ion-input>
        </ion-item>

        <ion-item>
            <ion-label stacked>Ammenities Code</ion-label>
            <ion-input type="text"></ion-input>
        </ion-item>

        <ion-item>
            <ion-label stacked>WiFi Password</ion-label>
            <ion-input type="text"></ion-input>
        </ion-item>

        <ion-item>
            <ion-label stacked>Phone Number</ion-label>
            <ion-input type="text"></ion-input>
        </ion-item>

        <ion-item>
            <ion-label stacked>Departure Date</ion-label>
            <ion-datetime displayFormat="DD/MM/YYYY"></ion-datetime>
        </ion-item>

        <ion-item>
            <ion-label stacked>Notes</ion-label>
```

```
            <ion-textarea type="text"></ion-textarea>
        </ion-item>

    </ion-list>
</ion-content>
```

第一个就是添加了navbar，你已经知道他是怎么工作的。我们也用了一个为页面创建了一个小小的页首区用于描述页面上干嘛用的，这只是一个纯粹的装饰品。然后我们在一个列表里面有大量的输入域。不是用的标准HTML语法：

```
<input type="text" />
```

我们使用的是Ionic提供的，但是接受的是同样的类型（你应该注意到了最后一个是）。我们提供了*stacked*属性，这个属性让在输入域上“堆栈”标签，但是Ionic默认给输入域提供的样式有很多。目前我们只是给输入域添加了标签，稍后回来我们会其他一些其他的东西这样我们就可以获取用户的输入。
同时这里我们也有用到，这是Ionic提供的内置的日期和时间选择器。
现在没有什么不同的事情会发生，我们还是先来创建页的类定义吧。
> 修改 **src/pages/camp-details/camp-details.ts** 如下：

```
import { Component } from '@angular/core';
import { NavController } from 'ionic-angular';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { Data } from '../../providers/data';

@Component({
    selector: 'page-camp-details',
    templateUrl: 'camp-details.html'
})
export class CampDetailsPage {

    constructor(public navCtrl: NavController, public formBuilder: FormBuilder,public
dataService: Data) {
    }

    saveForm(): void {
    }
}
```

这里跟之前不同的是我们从Angular导入了**FormBuilder**和**Validators**。就跟上面说的一样，我们将编辑稍后创建的输入域来做一些实际的事情，这就是Form Builder（和ControlGroup）用来介入的地方了。Form Builder允许你创建和管理表单，Vlidators可以附加到指定的输入域以确保输入的是一个有效值（即，如果需要输入的是邮件地址，手机号等）。我们稍后再深入。

我们也添加了一个*saveForm*函数稍后用来保存用户在表单中的输入值。现在，我们移步到My Details页，跟这个页面很像。

> 修改 **src/pages/my-details/my-details.html** 页面为如下：

```
<ion-header>
    <ion-navbar color="primary">
        <ion-title>
            <img src = "assets/images/logo.png" class="logo" />
        </ion-title>
    </ion-navbar>
</ion-header>

<ion-content padding>
    <ion-card>
        <ion-card-header>
        My Details
        </ion-card-header>

        <ion-card-content>
        Update this form with your details so you have an easy reference for
        later.
        </ion-card-content>
    </ion-card>

    <ion-list no-lines>
        <ion-item>
        <ion-label stacked>Car Registration</ion-label>
        <ion-input type="text"></ion-input>
        </ion-item>

        <ion-item>
        <ion-label stacked>Trailer Registration</ion-label>
        <ion-input type="text"></ion-input>
        </ion-item>

        <ion-item>
        <ion-label stacked>Trailer Dimensions</ion-label>
        <ion-input type="text"></ion-input>
        </ion-item>

        <ion-item>
        <ion-label stacked>Phone Number</ion-label>
        <ion-input type="text"></ion-input>
        </ion-item>

        <ion-item>
        <ion-label stacked>Notes</ion-label>
        <ion-textarea type="text"></ion-textarea>
        </ion-item>
    </ion-list>
</ion-content>
```

没啥好解释的，只是有几个不同的输入域，他跟Camp Detail模板简直就是一毛一样。直接去类定义吧。

> 修改 **src/pages/my-details/my-details.ts** 如下：

```
import { Component } from '@angular/core';
import { NavController } from 'ionic-angular';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { Data } from '../../providers/data';

@Component({
    selector: 'page-my-details',
    templateUrl: 'my-details.html'
})
export class MyDetailsPage {
    constructor(public navCtrl: NavController, public formBuilder: FormBuilder, public
 dataService: Data) {

    }
    saveForm(): void {
    }
}
```

同样，这个跟Camp Details页基本一毛一样的。为这个枯燥的结局道个歉，但是对于布局部分来讲，这就是全部了，下一刻我们将近距离观察Form Builder以及让我们让你合理的输入。

第四课：用户输入和表单

这节课我们来看Camp Details和My Details页的功能，也就是这些功能组成我们的输入功能。如果你已经完成了本书的其他应用的话，或者之前用过Ionic 2的输入域，你可能有这样用过*ngModel*：

```
<ion-input type="text" [(ngModel)]="myField"></ion-input>
```

这种写法给输入域设置了双向数据绑定，这些我在基础部分详细讨论过，本质上他就是他这个输入域的值捆绑到本页类定义的：

```
this.myField
```

如果你改变*this.myField*的值的话，将会映射到这个输入框来，如果输入框的值改变的话将会被映射到*this.myField*。对于管理用户输入来将，这是个非常简单的方法，但是如果有更多更复杂的输入域的时候，那么就需要使用**FormBuilder**了。
通过使用From Builder，不仅可以使代码更漂亮，还可以在“控制”每个输入框的同时带来更多强大的功能（如果看一眼Giflists应用的话，会看到我们用来控制文本域改变的定语，通过他可以做很多神奇的事情）。
我们也可以使用**Validators**与Form Builder写作，他允许我们捆绑一个“验证”到指定的输入域，这个验证将会检查输入是否允许（即，是否是正确的邮件格式）。
我们直接进入实现过程吧，更直观些。我们会先实现Camp Details页面的功能，然后烤制到My Details页。
> 修改 **src/pages/camp=details/camp-details.html** 的列表部分为如下：

```
<ion-list no-lines>
    <form [formGroup]="campDetailsForm" (change)="saveForm()">

    <ion-item>
    <ion-label stacked>Gate Access Code</ion-label>
    <ion-input formControlName="gateAccessCode"   type="text"></ion-input>
    </ion-item>

    <ion-item>
    <ion-label stacked>Ammenities Code</ion-label>
    <ion-input formControlName="ammenitiesCode"   type="text"></ion-input>
    </ion-item>

    <ion-item>
    <ion-label stacked>WiFi Password</ion-label>
    <ion-input formControlName="wifiPassword" type="text"></ion-input>
    </ion-item>

    <ion-item>
    <ion-label stacked>Phone Number</ion-label>
    <ion-input formControlName="phoneNumber" type="text"></ion-input>
    </ion-item>

    <ion-item>
    <ion-label stacked>Departure Date</ion-label>
    <ion-datetime formControlName="departure"   displayFormat="DD/MM/YYYY"></ion-datet
ime>
    </ion-item>

    <ion-item>
    <ion-label stacked>Notes</ion-label>
    <ion-textarea formControlName="notes" type="text"></ion-textarea>
    </ion-item>
    </form>
</ion-list>
```

这里的输入框都差不多只是有一些比较重要的区别。首先，我们将它们包装到一个form标签中：

```
<form [formGroup]="campDetailsForm" (change)="saveForm()">
```

我们定义了*formGroup*的值为*campDetailsForm*，这个值很快就会跟Form Builder一起使用。同时，我们也监听了*(change)*事件并在检测到的时候触发*saveForm*函数，意思就是用户改变和切换输入框的时候都会触发*saveForm*函数，但是不会在输入单个字符的时候触发（我们想要的话也可以做到）。通常对于表单而言我们都是监听他的*(submit)*事件并在其中处理数据，但是我们不想用户点击“Save”按钮或者其他类似操作来处理，我们只是希望用户在输入了一个新值的时候尽快存储起来。
另一个重要的变更时我们给每个输入框添加了*formControlName*，并给他指定了一个名字（跟

我们将要使用ngModel做的差不多）。再次，我们就快把他和Form Builder一起使用了。
现在，我们看一下类定义。首先，我们改一下构造器：**>** 修改 **src/pages/camp-details/camp-details.ts** 的构造器如下：

```
campDetailsForm: FormGroup;

constructor(public navCtrl: NavController, public formBuilder: FormBuilder,public data
Service: Data) {
    this.campDetailsForm = formBuilder.group({
        gateAccessCode: [''],
        ammenitiesCode: [''],
        wifiPassword: [''],
        phoneNumber: [''],
        departure: [''],
        notes: ['']
    });
}
```

由于我们在模板中已经将*formGroup*定义为*campDetailsForm*，我们这里可以指定一个新的Form Builder组。我们通过将之前指定给输入框的**formControlName**提供进来创建一个新的组。注意，我们提供了一个包含了空白字符串的数组，他用于作为输入框的初始值，例如：

```
gateAccessCode: ['54321']
```

这段代码会将*dateAccessCode*输入框的初始值设为‘54321’。你也可以像这样在这里提供一个验证器（validator）：

```
gateAccessCode: ['', Validators.required]
```

以上代码会把*gateAccessCode*变为必需的域。这就是设置表单的全部内容了，现在我们只要实现*saveForm*函数来。
> 修改 **src/pages/camp-details/camp-details.ts** 的 **saveForm** 函数：

```
saveForm(): void {
    let data = this.campDetailsForm.value;
    //this.dataService.setCampDetails(data);
}
```

注意：我们注释掉了对数据服务的调用时因为我们目前没有实现他，不然的话TypeScript会向我们抛出错误。
现在，我们可以在任何时候通过*this.campDetailsForm.value*获取表单的值。他将会返回一个对象包含了所有的值，也就是我们需要保存的数据。所以我们将这些数据传给数据服务去保存（现在还未实现）。

记住，*saveForm*函数在任何输入框改变的时候会触发，所以任何他们改变的时候我们读取这些值并保存他们。
现在我们只需要把这些变更映射到我们的My Details页。这些基本上一样的，我们我直接就复制粘贴了。不解释。
> 修改 **src/pages/my-details/my-details.html** 为如下：

```
<ion-header>
    <ion-navbar color="primary">
        <ion-title>
            <img src = "assets/images/logo.png" class="logo" />
        </ion-title>
    </ion-navbar>
</ion-header>

<ion-content padding>
    <ion-card>
        <ion-card-header>
        My Details
        </ion-card-header>
        <ion-card-content>
        Update this form with your details so you have an easy reference for
        later.
        </ion-card-content>
    </ion-card>

    <ion-list no-lines>
        <form [formGroup]="myDetailsForm" (change)="saveForm()">
        <ion-item>
        <ion-label stacked>Car Registration</ion-label>
        <ion-input formControlName="carRegistration" type="text"></ion-input>
        </ion-item>

        <ion-item>
        <ion-label stacked>Trailer Registration</ion-label>
        <ion-input formControlName="trailerRegistration" type="text"></ion-input>
        </ion-item>

        <ion-item>
        <ion-label stacked>Trailer Dimensions</ion-label>
        <ion-input formControlName="trailerDimensions" type="text"></ion-input>
        </ion-item>

        <ion-item>
        <ion-label stacked>Phone Number</ion-label>
        <ion-input formControlName="phoneNumber" type="text"></ion-input>
        </ion-item>

        <ion-item>
        <ion-label stacked>Notes</ion-label>
        <ion-textarea formControlName="notes" type="text"></ion-textarea>
        </ion-item>
        </form>
    </ion-list>
</ion-content>
```

> 修改 **src/pages/my-details/my-details.ts**的构造器如下：

```
myDetailsForm: FormGroup;

constructor(public navCtrl: NavController, public formBuilder: FormBuilder,public data
Service: Data) {
    this.myDetailsForm = formBuilder.group({
        carRegistration: [''],
        trailerRegistration: [''],
        trailerDimensions: [''],
        phoneNumber: [''],
        notes: ['']
    });
}
```

> 修改 **src/pages/my-details/my-details.ts**的 **saveForm**函数如下：

```
saveForm(): void {
    let data = this.myDetailsForm.value;
    //this.dataService.setMyDetails(data);
}
```

确实，表单不是世界上最刺激的东西（最起码大部分人会这么认为），但是对于移动应用来讲他是极度重要的组件之一，所以了解他们的工作方式非常重要。学会使用Form Builder可以让你的表单更好管理更强大，但是有时候，一个简单的*[(ngModel)]*就足够了。
在下一节课，我们将学习稍有有趣一点的东西，也稍微复杂一点：我们来实现Google Maps！

第五课：实现Google地图和地理定位

Google Maps和移动应用是绝配。Google Maps API本身就是一黑科技来的，当你和设备组合起来的时候以为这移动性，不是静止的，他打开了一扇新世界的大门。现今市面上有很多使用Google Maps武装起来的牛逼应用。
即使地图功能不是你的应用的核心功能，他们经常作为补充功能出现（例如在地图上显示一个商业地址）。
本课中我们将在Location页上实现Google Maps。我们实际上要做的事情如下：

- 展示一个地图
- 允许用户设置在地图上当前的位置
- 显示上一次在地图上标记的位置
- 激活导航给用户回到预设位置

在应用中设置Google Maps SDK以及使用他是非常简单的事情。你只需要简单的加载Google Maps SDK脚本，然后与其API交互。事情变得稍微有点复杂，因为我们有一个主要的问题：如果用户没有联网咋办？
如果用户因为没有联网而用不了地图是不合理的，但是如何友好的处理这个问题呢？我们不希望报错和调处应用（因为Google Maps SDK没有加载）或者甚至造成地图没法工作，我们需要考虑以下几点：

- 如果用户没有网络连接怎么办？
- 如果用户在开始没有联网但是后面又有了？
- 如果用话开始有联网但是后面又没有了？

为处理上以上的情景，我们的解决方案需要实现如下几点：

- 不直接加载Google Maps SDK，等待到有网络连接再加载
- 在网络断开的时候，禁用Google Maps功能
- 网络再次连上的时候，启用Google Maps功能

为了让代码更清晰，我们将抽象出大量的功能到早先生成的Google Maps提供者中。这样在别的应用中就能很简单的重用这些代码了。
注意：我们这节课中将会使用Google Maps JavaScript SDK，但是你需要知道你也可以通过Cordova插件使用它们的本机SDK：https://github.com/mapsplugin/cordova-plugin-googlemaps
这节课会变得很大，所以我们还是尽快开始吧。我们先从实现一个Connectivity服务开始，也就是我们早先生成的另一个提供者。

Connectivity服务

这将是一个用来检查网络连接的快速简单的服务。如果用的是真机的话，我们可以使用之前安装的network information插件（这个更准确），但是如果应用是通过普通浏览器运行的话，我们使用*onLine*属性来检查联网（没插件那么准确）。

> 修改 **src/providers/connectivity.ts** 为如下：

```
import { Injectable } from '@angular/core';
import { Network } from 'ionic-native';
import { Platform } from 'ionic-angular';

declare var Connection;

@Injectable()
export class Connectivity {

    onDevice: boolean;

    constructor(public platform: Platform){
        this.onDevice = this.platform.is('cordova');
    }

    isOnline(): boolean {
        if(this.onDevice && Network.connection){
            return Network.connection !== Connection.NONE;
        } else {
            return navigator.onLine;
        }
    }

    isOffline(): boolean {
        if(this.onDevice && Network.connection){
            return Network.connection === Connection.NONE;
        } else {
            return !navigator.onLine;
        }
    }
}
```

这个服务直白明了。我们创建了一个*onDevice*便用，然后通过*Platform*来检查应用是否运行在真机上。然后我们使用*onDevice*变量来检查我们是否要检查*navigator.connection.type*来确认使用哪种网络信息插件，或者只是检查*navigator.onLine*属性。

我们定义了两个函数*isOnline*和*isOffline*，这样我们在任何导入了此服务的类里都可以调用这两个方法。技术层面上，你只需要定义其中一个方法就可以了（如果isOnline返回false的话我们显然就已经知道是离线状态），但是我觉得用两个其实也蛮好的。

这就是这个服务的全部，我们还是接着高Google Maps 服务吧。

Google Maps 服务

这个服务将持有咱们地图功能的大部分逻辑。这是一个很大的服务所以我们先来创建一点骨架然后功能一个一个的去实现。

> 修改 **src/providers/google-maps.ts** 为如下：

```
import { Injectable } from '@angular/core';
import { Connectivity } from './connectivity';
import { Geolocation } from 'ionic-native';

declare var google;

@Injectable()
export class GoogleMaps {
    mapElement: any;
    pleaseConnect: any;
    map: any;
    mapInitialised: boolean = false;
    mapLoaded: any;
    mapLoadedObserver: any;
    currentMarker: any;
    apiKey: string;

    constructor(public connectivityService: Connectivity) {
    }
    init(mapElement: any, pleaseConnect: any): Promise<any> {
    }
    loadGoogleMaps(): Promise<any> {
    }
    initMap(): Promise<any> {
    }
    disableMap(): void {
    }
    enableMap(): void {
    }
    addConnectivityListeners(): void {
    }
    changeMarker(lat: number, lng: number): void {
    }
}
```

注意，我们把刚制作的Connectivity服务导入进来了，然后作为服务注入到了构造器（译者：原文可能有误，说是added it as a provider in the decorator）。我们也从**Ionic Native**中导入了**Geolocation**插件。我们也在导入语句后面添加了*declare var google* -- 这样TypeScript编译器就不会大逃到我们了。由于我们动态加载Google Maps SDK，编译器不知道*google*是什么，在我们不声明（declare）这个变量的情况下，会想我们抛出错误。

你应该也注意到了有些函数返回Promise。这是因为我们想追踪地图完成加载的时机，所有这些函数组成一条链（一个接一个的调用），所以实际上我们可以菊花链这些promise到最初的

那个，也就是在最初的那个中我们可以设置一个处理器来处理地图加载完成的之后的逻辑。
最后，我们创建了很多函数，我们一个一个实现并解释。
> 修改 **init** 函数如下：

```
init(mapElement: any, pleaseConnect: any): Promise<any> {
    this.mapElement = mapElement;
    this.pleaseConnect = pleaseConnect;
    return this.loadGoogleMaps();
}
```

我们可以在导入这个服务的地方随时调用*init*函数来触发地图的加载流程。我们简单的返回*loadGoogleMaps*函数，这样将会执行这个方法并返回他的结果（也就是一个Promise）。
由于我们会从**location.ts**调用这个函数，我们就可以传入早先用*@ViewChild*获取的*map*和*pleaseConnect*元素。我们这里接受他们作为参数，作为成员变量这样我们可以在类里面任何地方访问到他们。
现在，我们来实现*loadGoogleMaps*函数。 **>** 修改 **loadGoogleMaps** 函数如下：

```
loadGoogleMaps(): Promise<any> {
    return new Promise((resolve) => {
        if(typeof google == "undefined" || typeof google.maps == "undefined"){
            console.log("Google maps JavaScript needs to be loaded.");
            this.disableMap();

            if(this.connectivityService.isOnline()){
                window['mapInit'] = () => {
                    this.initMap().then(() => {
                        resolve(true);
                    });
                    this.enableMap();
                }

                let script = document.createElement("script");
                script.id = "googleMaps";
                if(this.apiKey){
                    script.src = 'http://maps.google.com/maps/api/js?key=' +this.apiKe
y + '&callback=mapInit';
                } else {
                    script.src = 'http://maps.google.com/maps/api/js?callback=mapInit'
;
                }
                document.body.appendChild(script);
            }
        }
        else {
            if(this.connectivityService.isOnline()){
                this.initMap();
                this.enableMap();
            }
            else {
                this.disableMap();
            }
        }
        this.addConnectivityListeners();
    });
}
```

这个函数看起来蛮复杂的，但实际上很直白。首先我们通过检查*google*和*google.maps*是否可用来检查Google Maps是否加载，因为如果这两个变量可用的话就意味这SDK已经加载好了。

如果SDK没有加载完成的话，我们将触发加载流程。由于SDK还没有加载完成，我们首先调用了*disableMap*函数，这个函数告诉用户地图目前不可用。然后我们通过connectivity服务来检查用户是否在线，如果在线的话我们通过给应用添加*script*元素来注入Google Maps SDK。注意，URL加入了一个*&callback=mapInit*。这孕育我们在应用完成Google Maps SDK的加载后触发一个函数，在当前应用中，我们在完成加载后调用的是*initMap*和*enableMap*函数（马上就实现）。注意，我们这是了另一个Promise这样一来我们可以等到*initMap*完成之后再返

回。
如果SDK已经紧挨在完成，那么我们检查用户是否在线。如果在线的话，我们初始化并激活地图，如果不在线的话那么禁用地图。
最后一行调用的函数*addConnectivityListener*函数稍后实现。这个函数会监听上线和离线时间，这样我们是到何时启用和禁用地图，当用户打开应用初始化的时候是离线状态想要加载SDK的时候也一样。
接下来是另一个函数。
> 修改 **initMap** 函数如下：

```
initMap(): Promise<any> {

    this.mapInitialised = true;

    return new Promise((resolve) => {
        Geolocation.getCurrentPosition().then((position) => {
            let latLng = new google.maps.LatLng(position.coords.latitude,position.coor
ds.longitude);

            let mapOptions = {
                center: latLng,
                zoom: 15,
                mapTypeId: google.maps.MapTypeId.ROADMAP
            }

            this.map = new google.maps.Map(this.mapElement, mapOptions);
            resolve(true);

        });
    });
}
```

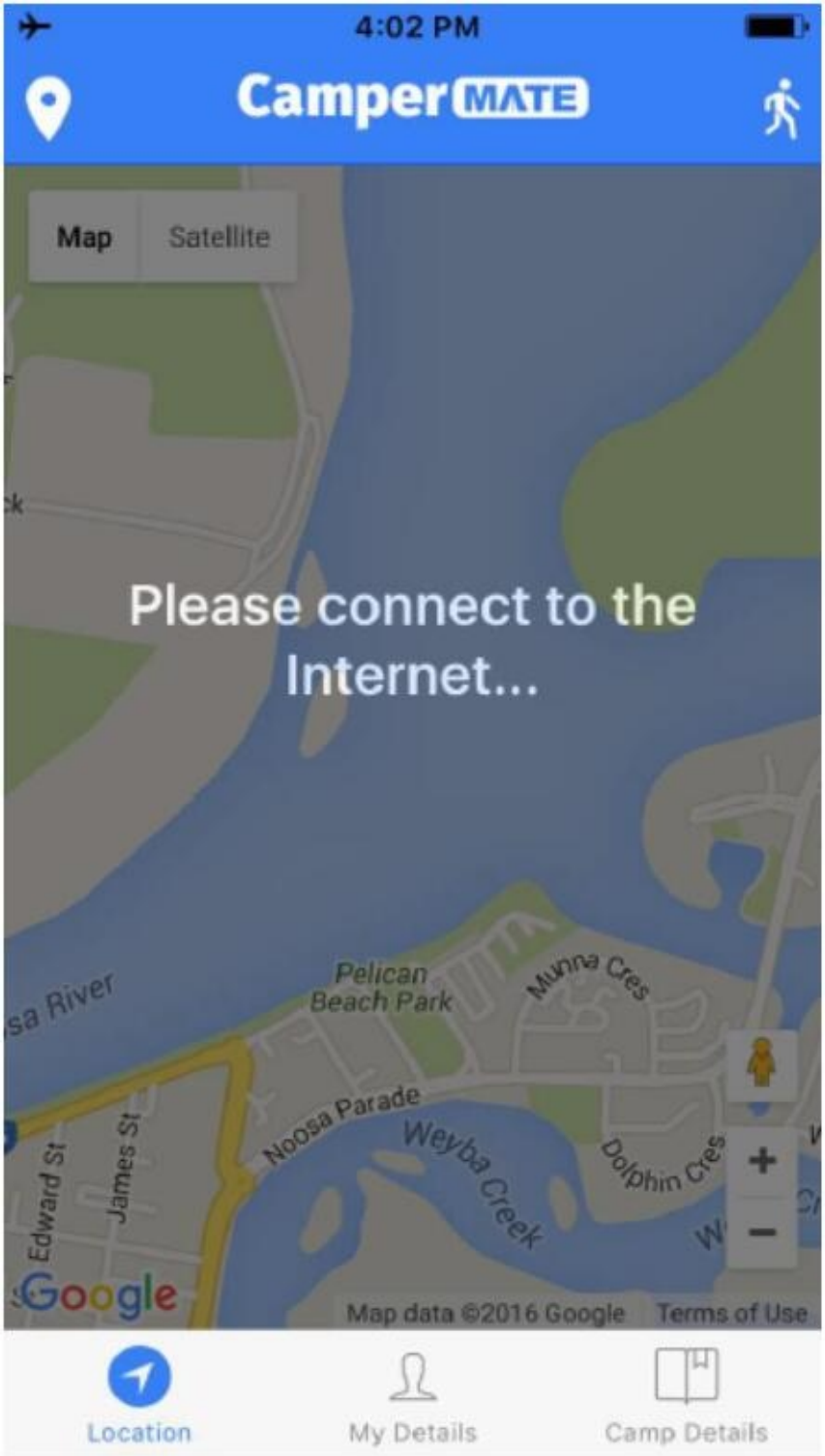
现在Google Maps SDK加载完成了，这个函数用于使用SDK设置一个新地图。我们想以用户当前位置来居中显示地图，我们我先调用了Geolocation插件的*getCurrentPosition*函数。一旦Promise返回解析完成，他传入的将是一个position对象，这个对象包含了用户当前的latitude和longitude。我们通过这些值，随同其他一些设置（缩放级别和地图类型）来创建一个新的地图实例。
这个地图将会被创建到传入的元素内（#map）。所以，这段代码运行后，Google Maps将会被添加到我们的Location页模板上。
此刻，地图已经准备好进行交互了，所以我们解析了promise链中的最后的promise，他将触发解析所有的promise，此时我们知道地图已经准备好了。
虽然我们已经加载完地图了，但是还需要创建一些函数。
> 修改 **disableMap** 和 **enableMap**函数如下：

```
disableMap(): void {
    if(this.pleaseConnect){
        this.pleaseConnect.style.display = "block";
    }
}

enableMap(): void {
    if(this.pleaseConnect){
        this.pleaseConnect.style.display = "none";
    }
}
```

我们这里做的是展示和隐藏Google Maps区域上的覆盖层，这样在用户没有连接到互联网的时候就不能使用地图，以及显示一个信息“Please connect to the Internet...”。以上代码用在在用户获得和失去网络连接的时候展示和隐藏一个信息元素。
我们需要给这个覆盖层元素进行自定义样式。
> 修改 **src/pages/location/location.scss** 为如下：

```
page-location {

    #please-connect {
        position: absolute;
        background-color: #000;
        opacity: 0.5;
        width: 100%;
        height: 100%;
        z-index: 1;
    }

    #please-connect p {
        color: #fff;
        font-weight: bold;
        text-align: center;
        position: relative;
        font-size: 1.6em;
        top: 30%;
    }
}
```

现在，当用户没有联网的时候，会看到这样的屏幕：



在达到这点之前我们还需要做一些事情。接下来我们来到了*addConnectivityListeners*函数。

> 修改 **src/providers/google-maps.ts** 的 **addConnectivityListeners** 函数如下：

```
addConnectivityListeners(): void {
    document.addEventListener('online', () => {
        console.log("online");
        setTimeout(() => {
            if(typeof google == "undefined" || typeof google.maps == "undefined"){
                this.loadGoogleMaps();
            }else {
                if(!this.mapInitialised){
                    this.initMap();
                }
                this.enableMap();
            }
        }, 2000);
    }, false);

    document.addEventListener('offline', () => {
        console.log("offline");
        this.disableMap();
    }, false);
}
```

如我所述，这个函数负责处理用户的联网状态在离线和上线之间的切换。我们在这里监听了‘online’和‘offline’事件，他们会在用户上线和离线的时候触发。
当用户上线的时候，我们检查Google Maps是否已经加载好了，如果没有的话就加载他。否则，检查地图是否初始化没有的话初始化他，有的话激活地图。注意，我们在这里有用到一个*setTimeout*，这些代码在用户上线2秒后运行一次 -- 之前即刻触发会遇到问题，所以我给连接一点点稳定下来的时间。
当用户离线的时候，我们禁用了地图。最后一个需要实现的函数是*changeMaker*函数。
> 修改 **changeMaker**函数如下：

```
changeMarker(lat: number, lng: number): void {
    let latLng = new google.maps.LatLng(lat, lng);
    let marker = new google.maps.Marker({
        map: this.map,
        animation: google.maps.Animation.DROP,
        position: latLng
    });

    if(this.currentMarker){
        this.currentMarker.setMap(null);
    }

    this.currentMarker = marker;
}
```

所有其他看书都很通用可以在其他任何要用到Google Maps的应用中直接使用，但是这个函数是这个应用才能使用的。在*addMarker*函数之上，这个函数会移除之前的marker添加一个新的marker。我们只希望用户设置一个露营地点。
我们只需要把需要添加marker的地方的latitude和longitude传入进去创建一个marker就可以了。如果有已经存在的marker的话我们就先移除他，然后添加刚创建的marker。
现在我们已经完成了Google Maps服务的设置，我们只要用它他就可以了！

实现Google Maps

我们已经做了大量的工作让地图正常工作，但是我们还一点点的路要走。现在我们要修改Location类定义来使用Google Maps服务。我们已经导入了这个服务，也将他添加到了providers数组以及给他创建了引用，这样我们就可以开始使用他了。
> 修改 **src/pages/location/location.ts** 的 **ionViewDidLoad** 函数：

```
ionViewDidLoad(): void {
    this.maps.init(this.mapElement.nativeElement,this.pleaseConnect.nativeElement).the
n(() => {
        //this.maps.changeMarker(this.latitude, this.longitude);
    });
}
```

首先，我们通过调用*this.maps.init*方法来触发地图的加载，这个函数将会返回一个promise。这个promise在地图完成加载的时候解析，当他解析完成的时候我们调用*changeMarker*函数。
现在是不会正常工作的因为latitude和longitude现在是undefined，所以我把它注释掉了。稍后，我们将将在保存在存储中的latitude和longitude。
这时候，地图应该加载完成了，且在屏幕上可见了，但是现在显示应该是错误的。首先，我们需要在*.scss*文件中添加一些样式来使他显示正常。
> 修改 **location.scss** 为如下：

```
page-location {
    #please-connect {
        position: absolute;
        background-color: #000;
        opacity: 0.5;
        width: 100%;
        height: 100%;
        z-index: 1;
    }

    #please-connect p {
        color: #fff;
        font-weight: bold;
        text-align: center;
        position: relative;
        font-size: 1.6em;
        top: 30%;
    }

    .scroll {
        height: 100%;
    }

    #map {
        width: 100%;
        height: 100%;
    }
}
```

接下来我们实现*setLocation*函数，这个函数用于将用户的露营设置为当前地点。
> 修改 **src/pages/location/location.ts** 的 **setLocation** 函数如下：

```
setLocation(): void {
    Geolocation.getCurrentPosition().then((position) => {
        this.latitude = position.coords.latitude;
        this.longitude = position.coords.longitude;

        this.maps.changeMarker(position.coords.latitude,   position.coords.longitude);
        let data = {
            latitude: this.latitude,
            longitude: this.longitude
        };

        //this.dataService.setLocation(data);
        let alert = this.alertCtrl.create({
            title: 'Location set!',
            subTitle: 'You can now find your way back to your camp site from  anywhere
 by clicking the button in the top right corner.',
            buttons: [{text: 'Ok'}]
        });

        alert.present();
    });
}
```

注意：再一次，我们注释掉了还未实现的数据服务。
在这里，我们用到了Geolocation来获取用户当前位置。一旦得到这些信息，我们将*this.latitude*和*this.longitude*改为用户当前位置，我们也用这个位置调用了*changeMarker*函数。我们也希望应用可以记住当前这个位置，所以我们为这个位置创建了一个对象传给数据服务去保存（记住，咱们还未实现此功能）。
一旦这些流程都完成了，我们出发一个警告框告诉用户位置设置成功。现在我们还剩下唯一的一个函数需要去定义。
> 修改 **src/pages/location/location.ts** 的 **takeMeHome** 函数如下：

```
takeMeHome(): void {
    if(!this.latitude || !this.longitude){
        let alert = this.alertCtrl.create({
            title: 'Nowhere to go!',
            subTitle: 'You need to set your camp location first. For now, want tolaunc
h Maps to find your own way home?',
            buttons: ['Ok']
        });
        alert.present();
    }
    else {
        let destination = this.latitude + ',' + this.longitude;
        if(this.platform.is('ios')){
            window.open('maps://?q=' + destination, '_system');
        } else {
            let label = encodeURI('My Campsite');
            window.open('geo:0,0?q=' + destination + '(' + label + ')','_system');
        }
    }
}
```

这个函数的作用是给用户展示他当前的位置和营地的位置。
首先，我们检查了*this.latitude*和*this.longitude*是否设置好了，如果没有的话我们会弹出警告框提醒他们需要先设置他们的地点。
如果已经设置好了的话，我们通过Google Maps和Apple Maps URL配置来启动应用，并且向他们提供坐标信息。如果我们运行在iOS上的话，那么我们会通过*maps://*配置来启动Apple Maps，如果我们是使用的Android应用的话，我们会通过*geo:*配置来启动Google Maps。现在，当用户触发此函数的时候，将会弹出一个地图给用户指示如何回到露营地点。
完成了！我们完成了我们的地图功能了。用户现在应该可以看到地图了，设置好他们的地址，触发回营方法。我们还有一个很重要的事情需要去处理，也就是用户回到应用的时候需要获取之前的保存的数据。下节课中，我们马上就处理这个事情，同时还有保存表单数据。

第六课：保存和取回数据

如果你认真学过之前的应用制作的话，那么制作一个Data服务对你来说相当简单了。这个跟之前的有一点点不同，因为之前的数据服务都是存储一套数据，但是这个应用中我们要存储很多数据：

- 营地坐标
- 营地细节
- 我的细节

但是理念还是差不多的，看起来可能会稍有不同。我们已经有了自己的表单并且我们的地图页也向咱们的Data服务发送数据了，所以我们要做的是保存它们即可，同时我们需要稍做变更来加载数据回应用。我们先从实现Data服务开始。
> 修改**src/providers/data.ts** 为如下：

```
import { Storage } from '@ionic/storage';
import { Injectable } from '@angular/core';

@Injectable()
export class Data {

    constructor(public storage: Storage){

    }

    setMyDetails(data: Object): void {
    let newData = JSON.stringify(data);
    this.storage.set('mydetails', newData);
    }

    setCampDetails(data: Object): void {
        let newData = JSON.stringify(data);
        this.storage.set('campdetails', newData);
    }

    setLocation(data: Object): void {
        let newData = JSON.stringify(data);
        this.storage.set('location', newData);
    }

    getMyDetails(): Promise<any> {
        return this.storage.get('mydetails');
    }

    getCampDetails(): Promise<any> {
        return this.storage.get('campdetails');
    }

    getLocation(): Promise<any> {
        return this.storage.get('location');
    }
}
```

Storage是Ionic的通用存储服务，他负责使用最佳存储方案的同时给我们提供了统一的API。当运行在设备上的时候，如果SQLite插件可用（早先安装过了），他会使用本机的SQLite数据库来存储数据。由于SQLite只会在应用运行在真机上才可用，*Storage*在SQLite不可用的情况下用*IndexedDB*，*WebSQL*，或者是浏览器的*localStorage*。
应当尽量使用SQLite，因为基于浏览器的本地存储不怎么可靠，可以被操作系统随机清理掉。自己的数据存在被随机清理掉明显体验很不好。
在其他应用中，我们基本只有两个函数，一个获取数据一个存储数据。在本应用中，由于我们涉及到三套数据，所以我们创建三个**set**函数和三个**get**很熟。set方法接受数据传入然后对传入的数据进行存储（在将他转换成一个JSON字符串之后），get负责将这些数据从数据库中获取回来。get函数将返回一个用于resolve返回数据的promise，而不是直接返回数据，所

以我们需要在想要用他的时候设置好处理器。
现在我们设置好了Data服务，我们只需要在重新打开应用的时候加载保存的数据即可。所以，现在我们对Location，MyDetails以及Camp Details页进行变更因为他们也需要载入数据。
我们先从Location页开始。
> 修改 **src/pages/location/location.ts** 的 **ionViewDidLoad** 如下：

```
ionViewDidLoad(): void {
    this.platform.ready().then(() => {
        this.dataService.getLocation().then((location) => {
            let savedLocation: any = false;

            if(location && typeof(location) != "undefined"){
                savedLocation = JSON.parse(location);
            }

            let mapLoaded = this.maps.init(this.mapElement.nativeElement,
            this.pleaseConnect.nativeElement).then(() => {
                if(savedLocation){
                    this.latitude = savedLocation.latitude;
                    this.longitude = savedLocation.longitude;

                    this.maps.changeMarker(this.latitude, this.longitude);
                }
            });
        });
    });
}
```

我们这里首先是从缓存里面获取用户位置，然后触发地图的加载。如果缓存中有位置信息的话，那么我们的*this.latitude*和*this.longitude*就需要用到这些数据，我们将在地图加载完成的时候将它们提供给*changeMaker*函数。如果没有存储位置的话我们加载地图就够了。
接下来我们处理Camp Details页。
> 修改 **src/pages/camp-details/camp-details.ts** 页为如下：

```
import { Component } from '@angular/core';
import { NavController, Platform } from 'ionic-angular';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { Data } from '../../providers/data';

@Component({
    selector: 'page-camp-details',
    templateUrl: 'camp-details.html'
})
export class CampDetailsPage {
    campDetailsForm: FormGroup;
    constructor(public navCtrl: NavController, public platform: Platform, public formB
uilder: FormBuilder, public dataService: Data) {
        this.campDetailsForm = formBuilder.group({
            gateAccessCode: [''],
            ammenitiesCode: [''],
            wifiPassword: [''],
            phoneNumber: [''],
            departure: [''],
            notes: ['']
        });
    }

    ionViewDidLoad(){
        this.platform.ready().then(() => {
            this.dataService.getCampDetails().then((details) => {
                let savedDetails: any = false;

                if(details && typeof(details) != "undefined"){
                    savedDetails = JSON.parse(details);
                }

                let formControls: any = this.campDetailsForm.controls;
                if(savedDetails){
                    formControls.gateAccessCode.setValue(savedDetails.gateAccessCode);
                    formControls.ammenitiesCode.setValue(savedDetails.ammenitiesCode);
                    formControls.wifiPassword.setValue(savedDetails.wifiPassword);
                    formControls.phoneNumber.setValue(savedDetails.phoneNumber);
                    formControls.departure.setValue(savedDetails.departure);
                    formControls.notes.setValue(savedDetails.notes);
                }
            });
        });
    }

    saveForm(): void {
        let data = this.campDetailsForm.value;
        this.dataService.setCampDetails(data);
    }
}
```

可以看到我们加上了从数据服务获取数据的调用然后在其中做了一些处理。记得之前我教你在使用Form Builder创建自己的组的时候怎么样去提供默认值的吧：

```
gateAccessCode: ['value here']
```

你也许很期待通过使用保存的值作为初始值来创建你自己的表单。不幸的是，他可能不能这么做。取回数据的过程是异步的，虽然数据取回非常的快但是还是有一个等待时间。而Form Builder组需要立刻创建，否则会报错。所以，我们是立刻创建Form Builder组，然后通过*this.campDetailsForm.controls*来获取他的音乐。然后，我们就可以通过每个空间的*setValue*方法来设置存储的值。
接着只需要对My Details页做同样的事情就可以了。
> 修改 **src/pages/my-details/my-details.ts** 为如下：

```
import { Component } from '@angular/core';
import { NavController, Platform } from 'ionic-angular';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { Data } from '../../providers/data';

@Component({
    selector: 'page-my-details',
    templateUrl: 'my-details.html'
})
export class MyDetailsPage {
    myDetailsForm: FormGroup;
    constructor(public nav: NavController, public platform: Platform, public formBuild
er: FormBuilder, public dataService: Data) {
        this.myDetailsForm = formBuilder.group({
            carRegistration: [''],
            trailerRegistration: [''],
            trailerDimensions: [''],
            phoneNumber: [''],
            notes: ['']
        });
    }

    ionViewDidLoad() {
        this.platform.ready().then(() => {
            this.dataService.getMyDetails().then((details) => {
                let savedDetails: any = false;

                if(details && typeof(details) != "undefined"){
                    savedDetails = JSON.parse(details);
                }

                let formControls: any = this.myDetailsForm.controls;
                if(savedDetails){
                    formControls.carRegistration.setValue(savedDetails.carRegistration
);
                    formControls.trailerRegistration.setValue(savedDetails.trailerRegi
stration);
                    formControls.trailerDimensions.setValue(savedDetails.trailerDimens
ions);
                    formControls.phoneNumber.setValue(savedDetails.phoneNumber);
                    formControls.notes.setValue(savedDetails.notes);
                }
            });
        });
    }

    saveForm(): void {
        let data = this.myDetailsForm.value;
        this.dataService.setMyDetails(data);
    }
}
```

在上面的 Camp Details和 My Details类中，我们都接触了数据服务调用的注释，我们也需要接触Location里面的数据服务调用的注释。

> 修改 **src/pages/location/location.ts** 的 **setLocation()** 函数：

```
setLocation(): void {
    Geolocation.getCurrentPosition().then((position) => {
        this.latitude = position.coords.latitude;
        this.longitude = position.coords.longitude;
        this.maps.changeMarker(position.coords.latitude, position.coords.longitude);

        let data = {
            latitude: this.latitude,
            longitude: this.longitude
        };

        this.dataService.setLocation(data);
        let alert = this.alertCtrl.create({
            title: 'Location set!',
            subTitle: 'You can now find your way back to your camp site from anywhere
by clicking the button in the top right corner.',
            buttons: [{text: 'Ok'}]
        });

        alert.present();
    });
}
```

这样就可以了！所有输入到应用里面的数据现在在应用重新加载的时候就可以保持原样了。我们已经快要来到应用制作的结尾了，显然，我们还要添加一些样式（尽管现在看起来中规中矩），但是下一节课我们要做的是添加一个额外的小功能到应用里。

第七课：重用组件

听说你喜欢app，那么我就在你的app里面放一个app好了。目前为止我们的应用有三个标签页了，我们现在要加个全新的标签页了。我们本书的第一个应用QuickList跟咱们应用很搭调呢。外出露营的时候Checklist非常方便的说。

我们这节课要做的是基上就是将Quick List功能“拖放”到当前应用中。在Ionic 2和Angular 2 的模块化的天性之下，这些事情可以很轻易的实现。如果你之前已经创建好了Quick List的话，你只要去其中拷贝代码即可，如果没有的话，那么去下载的包里拷贝代码。

没有像“拖放”那么容易，因为直接拖放的话会有一些冲突，但是还是很直白。我们先从文件拷贝开始。

> 拷贝 **QuickList** 的 **models** 文件夹到 **CamperMate** 的 **src** 文件夹中

> 拷贝 **QuickLists** 的 **checklist** 页文件夹到 **CamperMate**的 **pages** 文件夹

这两部非常简单，但是现在我们会遇上一点小问题。我们也想要从QuickLists中拷贝**HomePage**，但是我们的CamperMate已经有**HomePage**了，所以我们需要做一点点变通。

> 在**CamperMate** 的 **pages** 文件夹内创建一个新的文件夹名为 **quicklistshome**

> 将 **QuickLists** 的 **home**文件夹里面的内容（**home.ts,home.scss,home.html**）拷贝到刚才新建的 **quicklistshome** 里面

> 将 **home.ts,home.scss,home.html**重命名为**quicklistshome.ts,quicklistshome.scss, quicklistshome.html**

现在得到的结果是我们从QuickLists中将他的整个**home**页都拷贝过来了，但是拷贝过来之后我们都重命名为**quicklistshome**，因为我们不想让他和已有的home页有冲突。如果对上面的内容感觉比较迷惑，那么请查看本书源代码的文件夹结构。

由于我们对文件名进行了变更，我们也需要对代码进行一些变动。我们需要将QuickLists Home Page的类名，同时我们也要修改模板的名字：

> 对 **src/pages/quicklistshome/quicklistshome.ts** 进行如下变更：

```
import { Component } from '@angular/core';
import { NavController, AlertController, Platform } from 'ionic-angular';
import { ChecklistPage } from '../checklist/checklist';
import { ChecklistModel } from '../../models/checklist-model';
import { Keyboard } from 'ionic-native';
import { Data } from '../../providers/data';

@Component({
    selector: 'page-quicklistshome',
    templateUrl: 'quicklistshome.html'
})
export class QuickListsHomePage {
    checklists: ChecklistModel[] = [];
    local: Storage;

    constructor(public nav: NavController, public platform: Platform, public dataServi
ce: Data, public alertCtrl: AlertController) {
    }

    ionViewDidLoad(){
    this.platform.ready().then(() => {
        this.dataService.getData().then((checklists) => {
            let savedChecklists: any = false;

            if(checklists && typeof(checklists) != "undefined"){
                savedChecklists = JSON.parse(checklists);
            }

            if(savedChecklists){
                savedChecklists.forEach((savedChecklist) => {
                    let loadChecklist = new ChecklistModel(savedChecklist.title, saved
Checklist.items);
                    this.checklists.push(loadChecklist);

                    loadChecklist.checklist.subscribe(update => {
                        this.save();
                    });
                });
            }
        });
        });
    }
}
```

注意类型和templateUrl都变了。有一个我们忽略了的事情是我们没有将Quicklists应用的Data服务拷贝过来。这个因为我们的CamperMate应用已经有一个了，所以我们只要对他进行改动就可以了。

> 给 **src/providers/data.ts** 添加下面两个函数：

```
getData(): Promise<any> {
    return this.storage.get('checklists');
}

save(data: any): void {
    let saveData = [];
    //Remove observables
    data.forEach((checklist) => {
        saveData.push({
            title: checklist.title,
            items: checklist.items
        });
    });
    let newData = JSON.stringify(saveData);
    this.storage.set('checklists', newData);
}
```

这两个函数的命名方式和**data.ts**里面其他函数的命名方式不一样，保持这样的原因是避免对Quicklists功能做更多的修改。
样式方面没有多少要做的，但是我们的Quicklists应用的navbar都是用的*secondary*颜色，但是CamperMate用的是*primary*，所以我们需要对它进行改动。
> 修改 **src/pages/checklist/checklist.html** 的**navbar**为如下：

```
<ion-navbar color="primary">
```

> 修改 **src/pages/quicklistshome/quicklistshome.html** 的**navbar**为如下：

```
<ion-navbar color="primary">
```

> 修改 **src/pages/quicklistshome/quicklistshome.scss** 为如下：

```
page-quicklistshome {
    ion-item-sliding {
        margin: 5px;
    }

    .home-item {
        font-size: 1.2em;
        font-weight: bold;
        color: #282828;
        padding-top: 10px;
        padding-bottom: 10px;
    }

    .secondary-detail {
        display: block;
        color: #cecece;
        font-weight: 400;
        margin-top: 10px;
    }
}

button {
    border: none !important;
}
```

这样一来，我们就在CamperMate应用里面完美移植了QuickLists功能了。我们现在只要加上对于的标签栏就可以了。

> 修改 **src/pages/home/home.html**以包含下面第四个标签栏

```
<ion-tabs>
<ion-tab [root]="tab1Root" tabTitle="Location" tabIcon="navigate"></ion-tab>
<ion-tab [root]="tab2Root" tabTitle="My Details" tabIcon="person"></ion-tab>
<ion-tab [root]="tab3Root" tabTitle="Camp Details" tabIcon="bookmarks"></ion-tab>
<ion-tab [root]="tab4Root" tabTitle="Checklists" tabIcon="checkbox"></ion-tab>
</ion-tabs>
```

这样我们就可以去类定义中定义*tab4Root*了。

> 修改 **src/pages/home/home.ts**到如下：

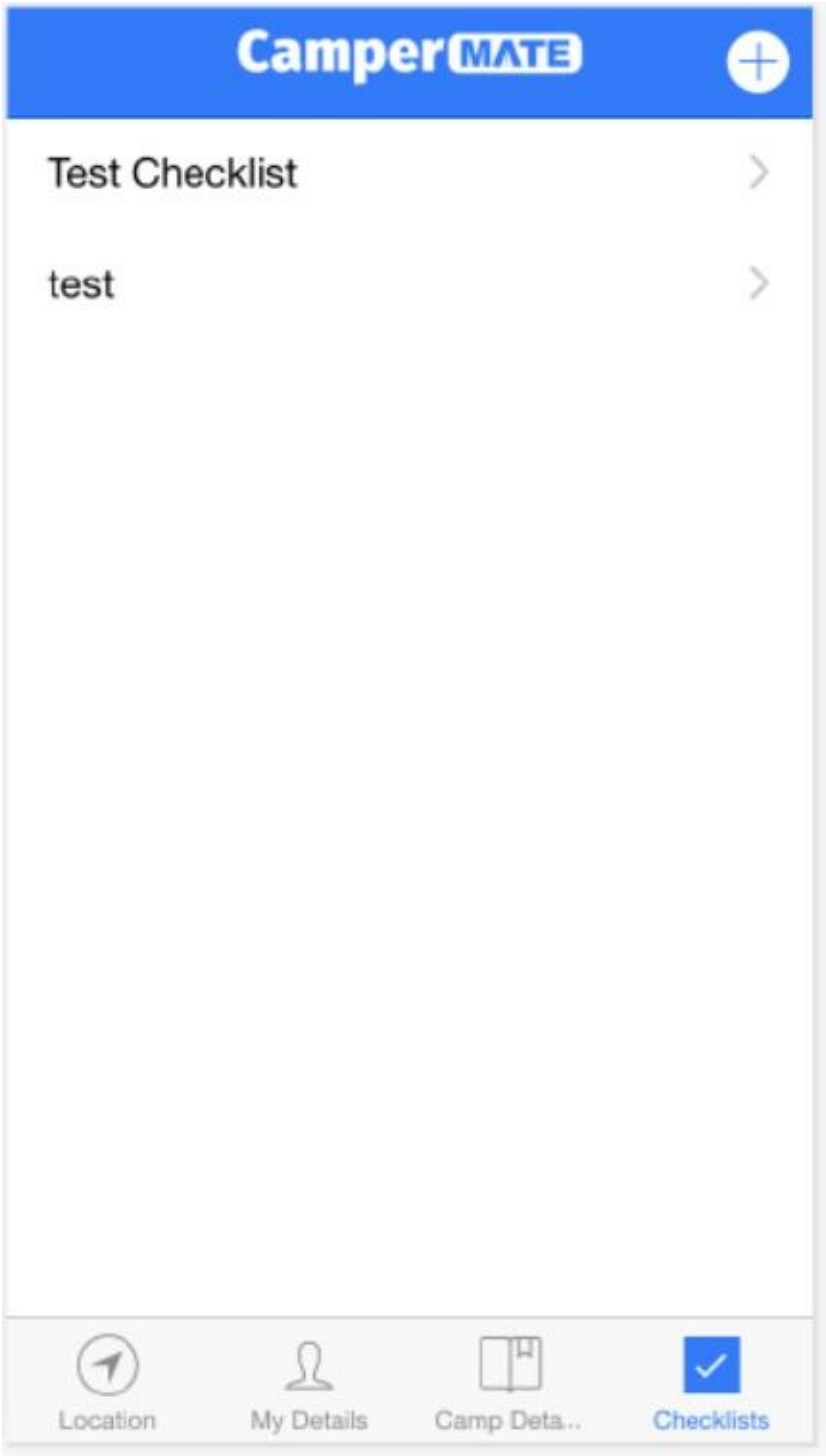
```
import { Component } from '@angular/core';
import { LocationPage } from '../location/location';
import { MyDetailsPage } from '../my-details/my-details';
import { CampDetailsPage } from '../camp-details/camp-details';
import { QuickListsHomePage } from '../quicklistshome/quicklistshome';

@Component({
    selector: 'page-home',
    templateUrl: 'home.html'
})
export class HomePage {

    tab1Root: any = LocationPage;
    tab2Root: any = MyDetailsPage;
    tab3Root: any = CampDetailsPage;
    tab4Root: any = QuickListsHomePage;

    constructor(){

    }
}
```

新加的页面也需要去**app.module.ts**中进行导入。

> 修改**src/app/app.module.ts** 为如下：

```
import { NgModule } from '@angular/core';
import { IonicApp, IonicModule } from 'ionic-angular';
import { MyApp } from './app.component';
import { Storage } from '@ionic/storage';
import { HomePage } from '../pages/home/home';
import { LocationPage } from '../pages/location/location';
import { MyDetailsPage } from '../pages/my-details/my-details';
import { CampDetailsPage } from '../pages/camp-details/camp-details';
import { QuickListsHomePage } from '../pages/quicklistshome/quicklistshome';
import { ChecklistPage } from '../pages/checklist/checklist';
import { GoogleMaps } from '../providers/google-maps';
import { Connectivity } from '../providers/connectivity';
import { Data } from '../providers/data';

@NgModule({
    declarations: [
        MyApp,
        HomePage,
        LocationPage,
        MyDetailsPage,
        CampDetailsPage,
        QuickListsHomePage,
        ChecklistPage
    ],
    imports: [
        IonicModule.forRoot(MyApp)
    ],
    bootstrap: [IonicApp],
    entryComponents: [
        MyApp,
        HomePage,
        LocationPage,
        MyDetailsPage,
        CampDetailsPage,
        QuickListsHomePage,
        ChecklistPage
    ],
    providers: [Storage, Data, GoogleMaps, Connectivity]
})
export class AppModule {}
```

现在加载应用的话，应该可以看到一个新的标签页：



如你所见，这样模组是项目结构使得重用其他应用中的代码非常简单，如果用同名组件的话就会更简单了。
我们的应用现在看起来已经很好了，但是下节课我们要加入一点样式让他更好看些。

第八课：自定义样式

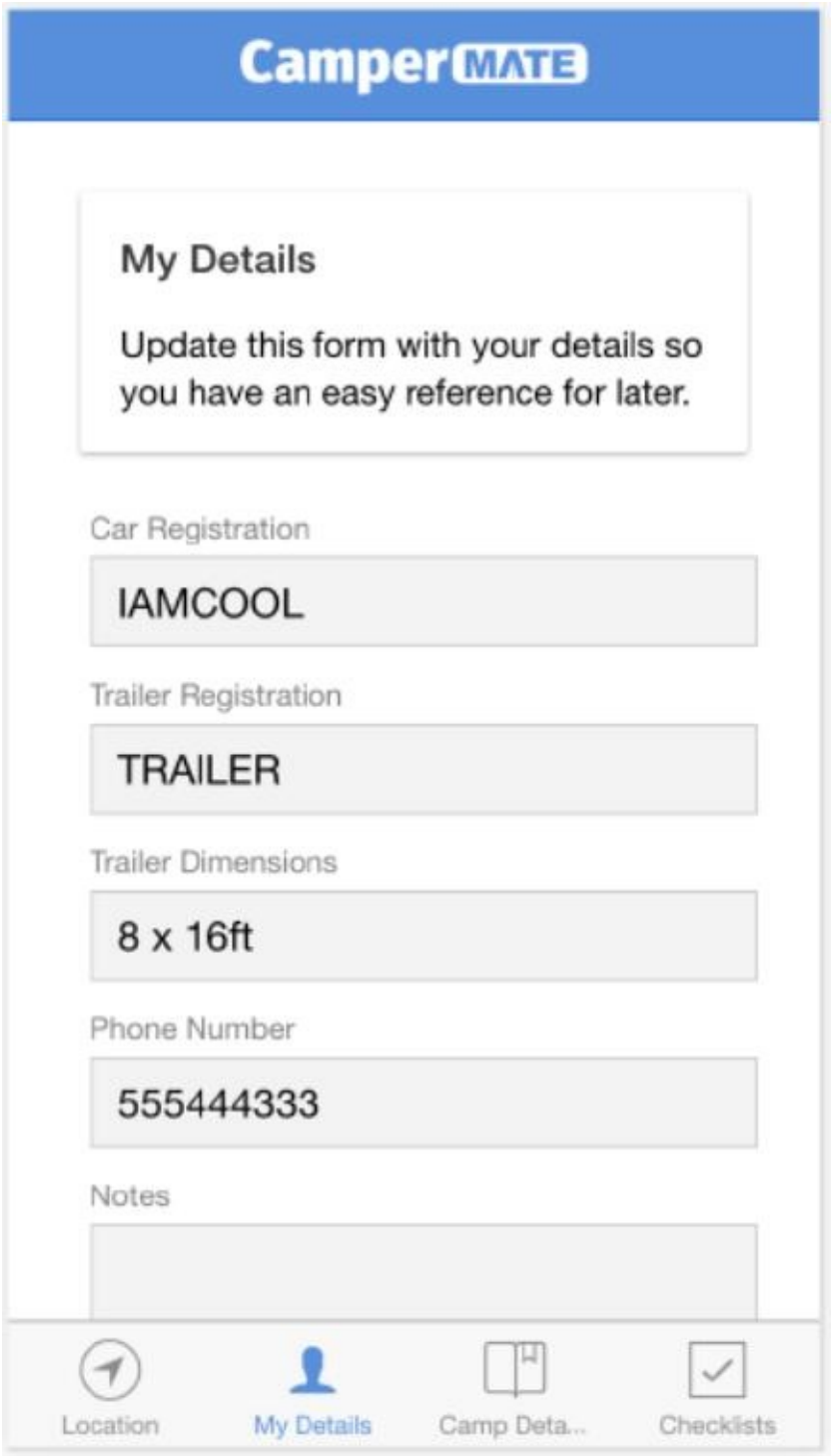
这个应用很漂亮了，所以这里没有太多自定义的东西，所以这节课非常的快。但是我们还是要做点啥。
我们先从修改颜色变量开始。
> 修改 **theme/variables.scss** 里面的命名颜色变量如下：

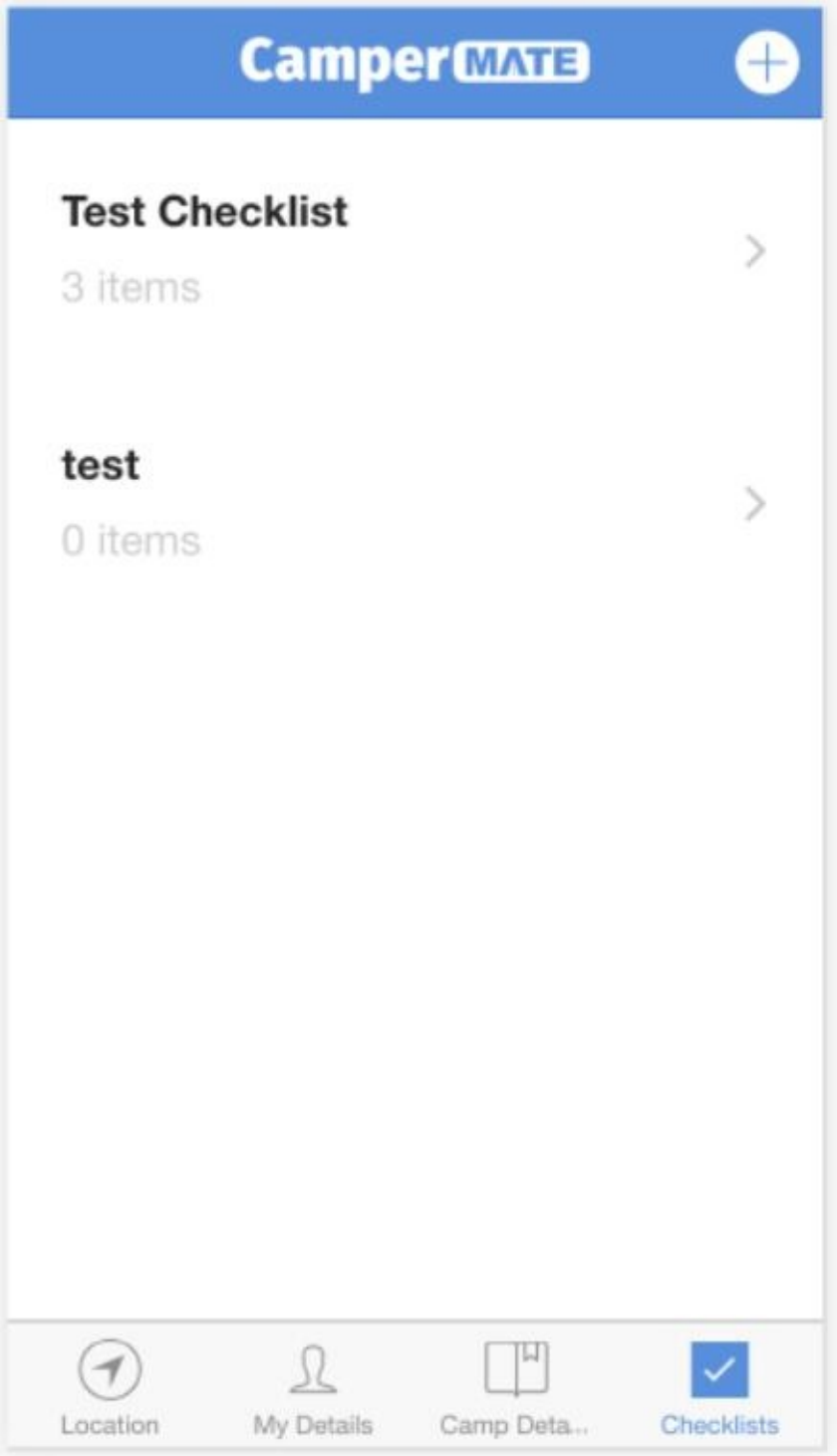
```
$colors: (
    primary: #5b91da,
    secondary: #32db64,
    danger: #f53d3d,
    light: #f4f4f4,
    dark: #222,
    favorite: #69BB7B
);
```

然后给应用的添加一些通用样式。
> 在**src/app/app.scss**中添加如下样式：

```
.logo {
    max-height: 39px;
    margin-top: 6px;
}

ion-input, ion-textarea {
    background-color: #F3F3F3;
    border: 1px solid #cecece;
    padding-left: 10px;
}

ion-textarea {
    height: 200px;
}

textarea {
    height: 180px;
}

button {
    border: none !important;
}
```

这里我们想要做到的主要事情是给我们的输入框添加一些样式。在这之前，输入框都是白色的，他的背景也是，所以你基本看不到输入框中哪里。现在他们的背景色灰色的，以及他有一些小小的额外的样式。我们给textarea添加了一点点样式来扩展他的默认高度，也覆盖了一个目前存在的bug，也就是textarea不会随着用户输入去调整。
现在应用看起来应该是这样的：





这是整本书里面最小的课程了，但是，我们完成了！CamperMate现在全部完成了。

结论

恭喜你完成了Camper Mate的制作教程。在开发过程中，我们学习到了很多东西，主要是以下几个：

- 创建表单和获取用户输入
- 实现Google Maps和创建自定义的provider用来操作他
- 创建标签页布局
- 保存和获取不同数据
- 重用其他应用的组件

改进的空间永远都存在，特别是当你学习事物的时候。遵循指导手册固然很好，但是自己去学习弄清楚一些事情就更完美了。希望你有足够的背景知识来自己完成一些功能扩展，以下是一些想法：

- 给Phone Number和Date输入域应用Validators以确保输入有效数据【中等】
- 在用户出发前一天使用本地通知（Snapaday应用中用到）提醒用户【困难】
- 添加一个域名为'Camp Details'允许用户拍照在此展示（参考Snapaday）【困难】

记住，在你学习的过程中，Ionic 2文档是你最好的朋友。

接下来？

我们已经有一个完整的应用了，但是这不是故事的结局。你需要让他运行到真实设备上去，提交到应用商店，这不是个简单的任务。本书的最后部分会带你完成这些内容。你可以先看看。

项目：露营聊天软件（Camper Chat）

第一课：介绍

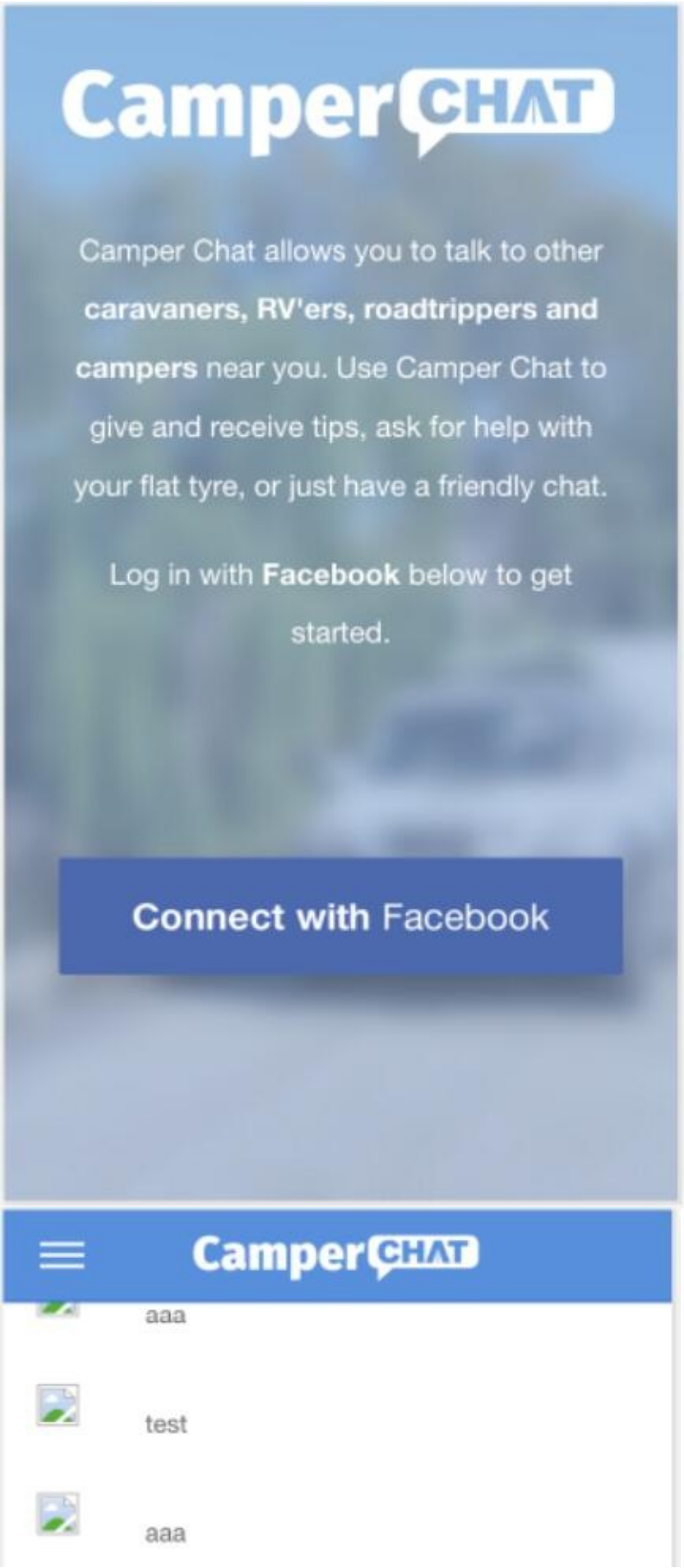
这个应用是很大个的，是“一个挑战”，是“万中选一”，是我们本书制作的最后一个应用，我觉得这个应用很适合用来作为结尾。相对与之前应用来讲，这个应用不需要完成之前的应用作为前提，因为这个应用所有东西都会详细解释，即使是之前应用中已经解释过了，但是这个应用的难度也上了一个台阶，我不会讲太多的基础知识。所以，如果你现在还是对Ionic 2不大舒服的话，你还是先熟悉一下其他的应用吧，因为他们简单些。
本部分我们将制作Camper Chat，实际上就是一个实时聊天应用。用户可以使用他们的Facebook账号登录，然后跟其他使用此应用的用户聊大篷车和露营的事情。这个应用最酷的地方是整合PouchDB来存储本地数据，以及使用Cloudant来同步PouchDB数据到远程后端。这意味着用户在离线的时候，他的数据也可以访问到，当用户在此上线的时候，会从远程后端获取最新数据。
为给你一个准确的定义，本应用的具体功能点如下：

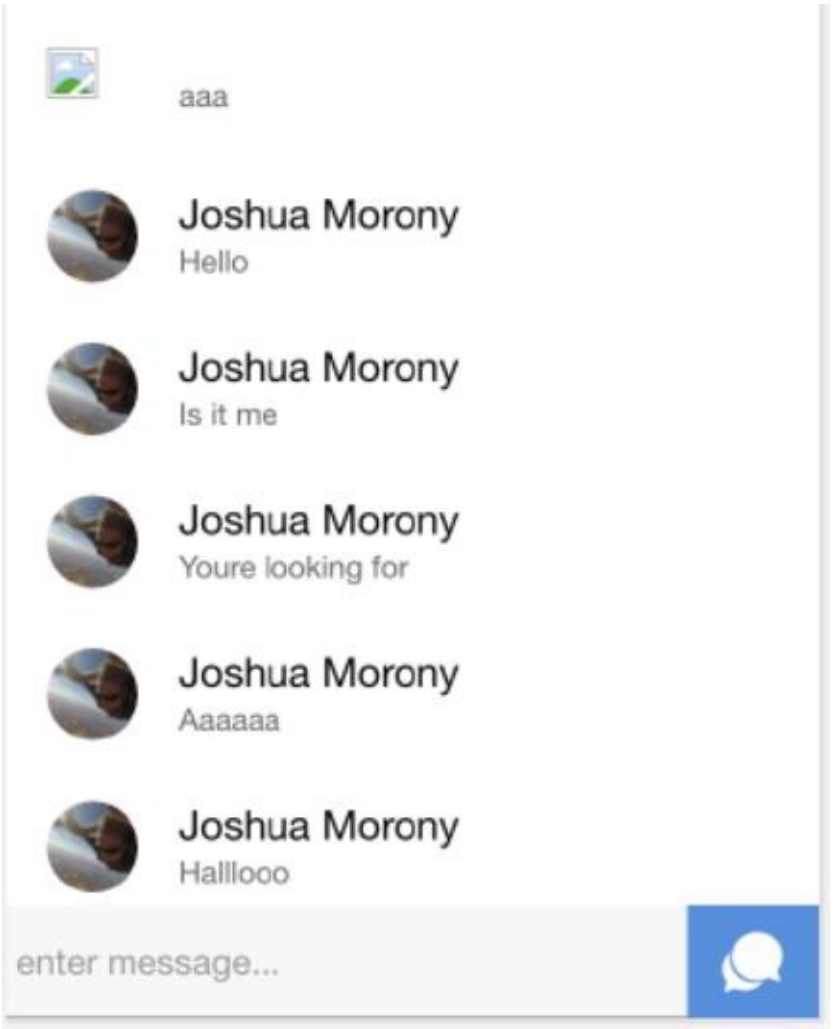
- 用户可以通过使用Facebook账号登录了来使用此应用
- 用户可以留言其他用户可以看到和回复（实时）
- 用户将在此应用中使用Facebook头像和名字
- 用户可以登出
- 用户可以查看‘About’页

有了以上需求定义，我们将要学习的知识点包括：

- 导航
- 使用侧边滑动菜单
- 使用PouchDB来存储本地数据
- 使用Cloudant来存储远程数据
- 使用Facebook API进行认证和其他功能
- 实时更新和显示数据

以下是一些应用截图给你预先一个参考：





课程结构

1. 准备工作
2. 登录页和侧滑菜单布局
3. 使用Facebook进行验证
4. 制作和展示信息与导航
5. 使用PouchDB和Cloudant制作本地和远程后端
6. 自定义样式

准备好了吗？
现在你知道了你要做什么，那么我们就可以开始了。

第二课：准备工作

本课是在旅程继续进行之前的一些准备工作。我们要生成应用，设置所有组件和需要用到的Cordova插件。完成本课之后我们应该有一个万事俱备的项目骨架，可以接着进行编码工作。开始新项目的第一准则是确保使用的是最新版的Ionic和Cordova，如果最近没有更新过的话可以运行如下命令：

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

或者

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

如果在安装Ionic或者生产新项目的时候遇到任何问题的话，检查下是否安装了最新版的Node。安装完之后，再次安装ionic之前请运行：

```
npm uninstall -g ionic npm cache clean
```

生成新应用

本应用将使用空白初始模板，跟名字说的一样，就是个空的Ionic项目。但是会有一个内置的页面名为**home**，我们下节课中将用作列表显示页。
> 运行如下命令生成新项目：

```
ionic start camperchat blank --v2
```

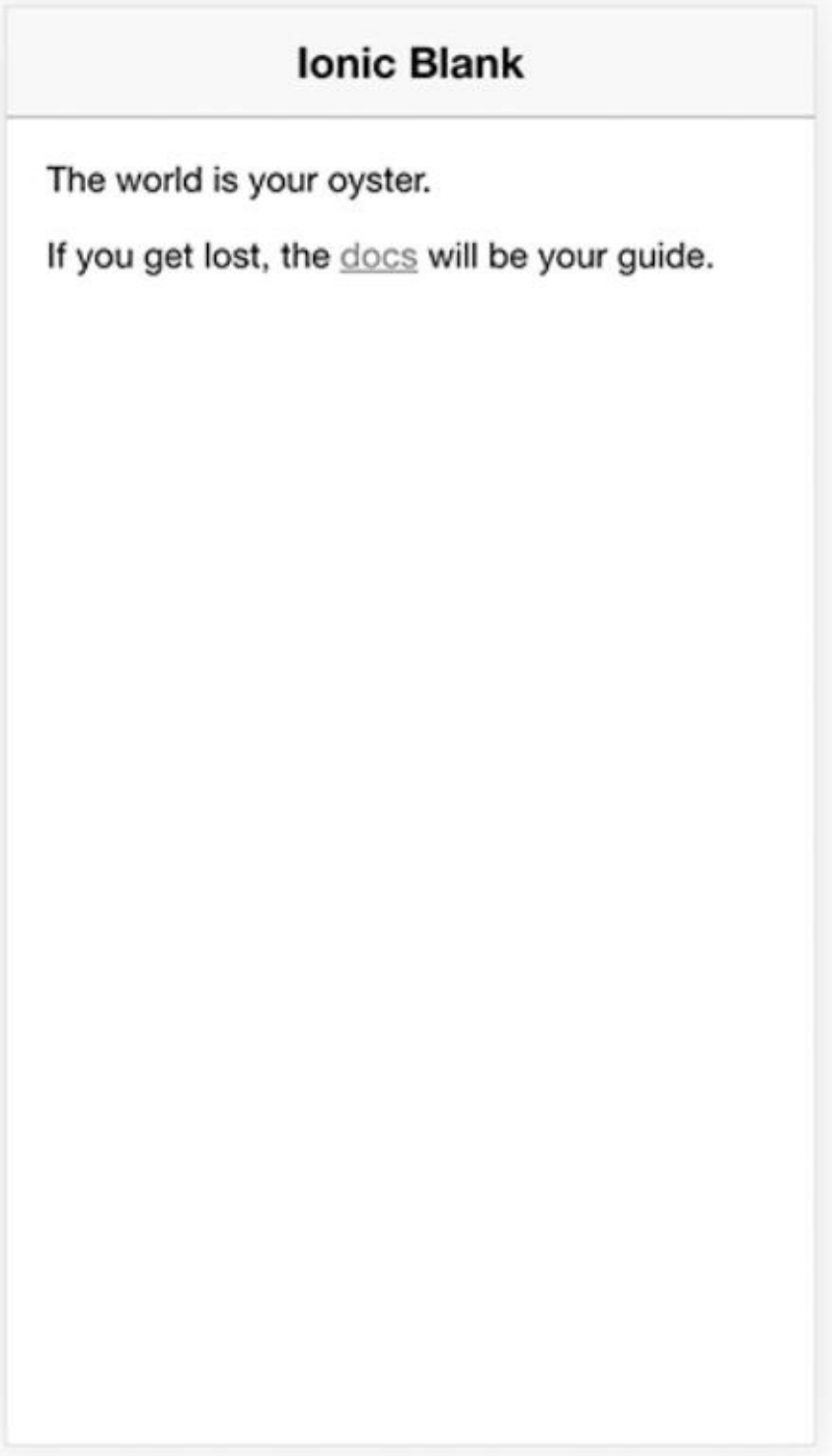
> 运行如下命令将新生成的项目作为当前目录：

```
cd camperchat
```

现在可以在你中意的编辑器中打开这个项目了。通过以下命令可以预览创建的应用：

```
ionic serve
```

看起来是这样的：



创建需要的组件

这个应用会有几个页面，我们需要用到登录页，主页，关于页。主页已经自动生成，我们只需要创建登录页和关于页就可以了。

> 运行如下命令生成 **Login** 页面：

```
ionic g page Login
```

> 运行如下命令生成 **About** 页面：

```
ionic g page About
```

创建需求的服务

跟标签页一样，我们同时也需要去创建一些服务。我们将创建一个数据服务来保存和获取信息数据，以及一些从Facebook API获取的数据。
> 运行如下命令生成 **Data** 提供者：

```
ionic g provider Data
```

将页面和服务添加到App Module

为了能够在项目里面可以使用这些页面和服务，我们需要将它们添加到**app.module.ts**文件里。所有我们自己创建的页面都需要添加到*declarations*数组和*entryComponents*数组里，所有我们创建的数据提供者都需要添加到*providers*数组，其他自定义组件或者管道（pipe）只需要添加到*declarations*数组即可。我们的数据模型只是一个简单的类，我们需要在任何地方使用，所以不用在模组里面设置。

> 修改**src/app/app.module.ts**到以下：

```
import { NgModule } from '@angular/core';
import { IonicApp, IonicModule } from 'ionic-angular';
import { MyApp } from './app.component';
import { HomePage } from '../pages/home/home';
import { LoginPage } from '../pages/login/login';
import { AboutPage } from '../pages/about/about';
import { Data } from '../providers/data';

@NgModule({
    declarations: [
        MyApp,
        HomePage,
        LoginPage,
        AboutPage
    ],
    imports: [
        IonicModule.forRoot(MyApp)
    ],
    bootstrap: [IonicApp],
    entryComponents: [
        MyApp,
        HomePage,
        LoginPage,
        AboutPage
    ],
    providers: [Data]
})
export class AppModule {}
```

添加需要的平台

在给指定平台制作应用之前，你需要将它们添加到你的项目。

> 运行以下命令添加**iOS**平台：

```
ionic platform add ios
```

> 运行以下命令添加**Android**平台：

```
ionic platform add android
```

安装**PouchDB**

由于我们会在项目中用到PouchDB，所以我们需要先安装他。跟安装Ionic Native一样，你只需要简单的运行如下命令就可以了：

```
npm install pouchdb --save
```

我们也要给PouchDB安装“typings”这样TypeScript编译器就不会抱怨了（因为他不知道PouchDB是什么）。
> 运行如下命令按钮**PouchDB**的**types**

```
npm install @types/pouchdb --save --save-exact
```

添加需要的**Cordova**插件

这个应用将会用到不同的Cordova插件。记住，Cordova插件只能在真实设备上运行。我将在添加他们的时候解释。
> 运行以下命令添加**SQLite**插件：

```
ionic plugin add cordova-sqlite-storage
```

这个插件让你可以访问本地存储SQLite数据库。我们在此应用中添加他的原因是Ionic本地存储服务可以使用插件提供的稳定输出存储。
> 运行以下命令添加**App Browser**插件：

```
ionic plugin add cordova-plugin-inappbrowser
```

这个插件让我们可以启动外部网站。我们会用这个插件来给Facebook插件使用。添加Facebook插件本事实际上需要做更多的工作而不是运行一个命令就够了的，所以我们稍后在做这个。 **>** 运行以下命令添加**Status Bar**插件：

```
ionic plugin add cordova-plugin-statusbar
```

我们给所有项目添加此插件用来在应用中控制状态栏（设备屏幕顶部的状态条，包括时间，电池信息等等）。

> 运行以下命令添加**Splash Screen**插件：

```
ionic plugin add cordova-plugin-splashscreen
```

此插件允许我们控制闪屏（打开应用的时候的全屏画面）。

> 运行以下命令添加**Keyboard**插件：

```
ionic plugin add ionic-plugin-keyboard
```

这个插件允许我们控制软键盘。

> 运行以下命令添加**Whitelist**插件：

```
ionic plugin add cordova-plugin-whitelist
```

所有应用会用到这个插件，他定义了应用里可以加载什么样的资源。没有他的话，你尝试加载的资源都会不成功。

添加了这个插件后，你也需要到**index.html**中定一个一个“Content Security Policy”。我们将添加一个非常宽松的策略实际上允许我们加载任何资源。基于你的应用，你可以提供一个更严格的策略，但是对于开发而言开放性策略就可以了。

> 修改 **src/index.html**文件，添加一下**meta**标签：

```
<meta http-equiv="Content-Security-Policy" content="font-src 'self' data:;
img-src * data:; default-src gap://ready file://* *; script-src 'self'
'unsafe-inline' 'unsafe-eval' * ; style-src 'self' 'unsafe-inline' *">
```

> 运行以下命令添加**Crosswalk**插件：

```
ionic plugin add cordova-plugin-crosswalk-webview
```

这个另一个每个应用都要添加的插件，但是你也可以先不添加。添加了这个插件后，在你编译Android的时候将会使用“Crosswalk”。Android有很多问题，特别是老设备，因为有太多不同的团建版本，不同的版本有不同的浏览器（记住，鉴于我们是制作HTML5应用，他实际上就是一个搭载的浏览器用来运行我们的应用）。Crosswalk做的是将一个现代的浏览器打包到应用中，这样一来应用无论是运行在什么设备上，都会使用相同的浏览器来运行，并且Crosswalk浏览器可以很好改善执行效率。

唯一的不足之处就是你的应用尺寸明显的变大了很多。总体上，我觉得这很值得，我也建议你使用他，如果你接受不了的话，也可以不用。更多关于Crosswalk Project的信息，请参考网站：https://crosswalk-project.org/

设置图片

制作此应用的时候，会用到一些图片。你下载的包里面已经包含了这些图片，但是你需要去生成的项目里面设置好他们。

> 将下载包 **src/assets**文件夹下面的**images**文件夹复制到应用里的 **src/assets**下面

总结

就这样！我们设置好了，准备好继续前几，现在我们开始进入到有趣的部分了。

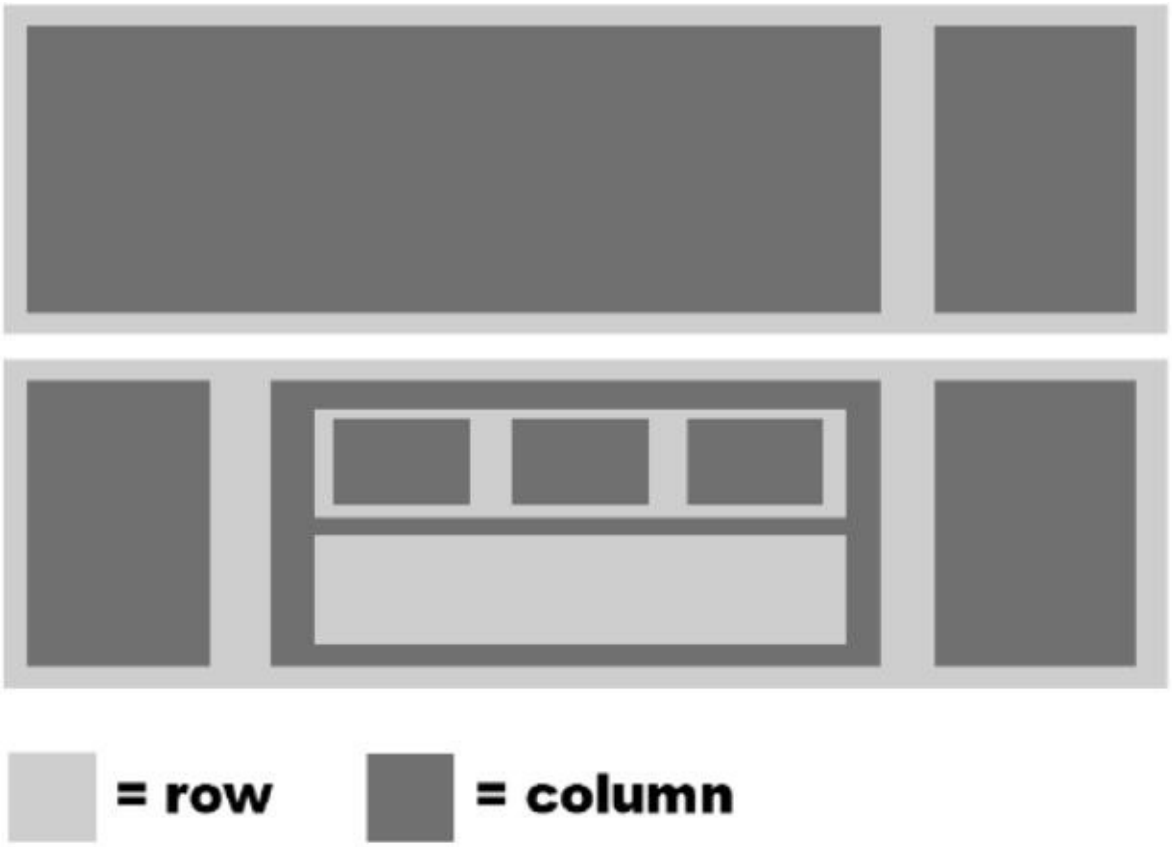
第三课：登录页面和滑动菜单布局

这个应用中我们将要实现一个实际的普通页面工作流，具体讲就是，一个登录页引导至主应用。在大部分需要用户登录验证的应用中，都会首先显示验证屏幕，只有在用户登录成功之后才会来到主应用。
在我们这个应用中，我们会展示一个登录页，其中有一个使用Facebook登录的按钮，这个按钮会导向一个带有侧滑菜单布局的页面。咱们的登录页非常简单（至少目前是这样的）所以我们开始吧。
> 修改 **src/pages/login/login.html**为如下：

```
<ion-content padding>
    <ion-row>
        <ion-col>
        <img src="assets/images/logo.png" />
        </ion-col>
    </ion-row>
    <ion-row>
        <ion-col>
        <p>Camper Chat allows you to talk to other <strong>caravaners,
        RV'ers, roadtrippers and campers</strong> near you. Use
        Camper Chat to give and receive tips, ask for help with your
        flat tyre, or just have a friendly chat.</p>
        <p>Log in with <strong>Facebook</strong> below to get started.</p>
        </ion-col>
    </ion-row>
    <ion-row>
        <ion-col>
        <a (click)="login()"><img src="assets/images/facebook-button.png"/></a>
        </ion-col>
    </ion-row>
</ion-content>
```

这样就定义好了我们的登录页模板，如果这不是你制作的第一个Ionic 2应用的话（我当然也希望不是），你会注意到，我没有在模板中加入navbar。我们不需要显示标题，不需要显示返回按钮，或者类似的东西，所以不用navabar是相当正确的。
我们这里没做啥疯狂的事情，我们在这里用到了Ionic的栅格系统，也就是行与列来设置我们

的布局。当使用格子系统的和的时候，行都是上下排列，行中的列是左右排列的。下面这个图标可以帮你很好的说明：



这是个很简单的用力，因为我们只是使用格子来上下排列展示三个不同的部分，你可以在行中间插入多列甚至可以定义每个列的宽度：

```
<ion-row>
    <ion-col width-10></ion-col>
    <ion-col width-50></ion-col>
</ion-row>
```

同时我们也给Facebook登录图片添加了一个*(click)*处理器。这个*login*函数实际上是触发使用Facebook进行验证，但是目前我们只是将要触发一个页面切换到主页以便完成我们的布局设置。我们现在来设置他的类以便设置好页面切换。

> 修改 **src/pages/login/login.ts**为如下：

```
import { Component } from '@angular/core';
import { Platform, NavController, MenuController, AlertController } from 'ionic-angula
r';
import { Facebook } from 'ionic-native';
import { HomePage } from '../home/home';
import { Data } from '../../providers/data';

@Component({
    selector: 'page-login',
    templateUrl: 'login.html'
})
export class LoginPage {
    constructor(public nav: NavController, public platform: Platform, public menu: Men
uController, public dataService: Data, public alertCtrl:AlertController) {
        this.menu.enable(false);
    }

    login(): void {
        this.getProfile();
    }

    getProfile(): void {
        this.menu.enable(true);
        this.nav.setRoot(HomePage);
    }
}
```

可以看到，上面代码中，我们注入并设置好了NavController我们在调用login函数的时候用它来将*rootPage*设置为HomePage。这些代码都是通过*getProfile()*运行的，因为我们稍后会在用户通过Facebook认证后获取回去一些信息，但是在我们把用户发配到下一页之前，由于我们还没有实现这个功能，所以我们只是让代码把我们带到成功的场景。

MenuController用于与即将添加的侧滑菜单交互。稍后会详细讨论这个，但是我们不想这个菜单中登录页是展示，所以我们使用了菜单的*enable*函数在此页面上禁用此菜单。我们在切换到Home也之后我们再次启用他。

我们也从Ionic Native导入了Facebook插件，稍有将会是一。记住，使用一个Cordova插件不需要导入任何东西，甚至是Facebook插件，但是，如果你是通过Ionive Native使用的话那么你就得去导入他了。

我们想要登录页第一个展示，但是如果此时你看**app.module.ts**的时候，会发现当前默认的根页面是主页：

```
rootPage: any = HomePage;
```

所以，我们当然得修改一下。

> 修改 **src/app/app.component.ts**为如下：

```
import { Component, ViewChild } from "@angular/core";
import { Facebook } from 'ionic-native';
import { Nav, Platform, MenuController } from 'ionic-angular';
import { StatusBar } from 'ionic-native';
import { HomePage } from '../pages/home/home';
import { AboutPage } from '../pages/about/about';
import { LoginPage } from '../pages/login/login';
import { Data } from '../providers/data';

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

    rootPage: any = LoginPage;
    homePage: any = HomePage;
    aboutPage: any = AboutPage;

    constructor(public platform: Platform, public dataService: Data, public menu: Menu
Controller) {
        platform.ready().then(() => {
        });
    }

    openPage(page): void {
    }

    logout(): void {
    }
}
```

你也看到了，我们把根页设为了登录页，当然我们也得先导入Login也。我们现在也做了不少东西了。我们导入一个设置了Data provider的引用，我们后续会用他来保存和访问数据；Menu Controller，允许我们和创建中的侧滑菜单交互。同时，我们也用到了*@ViewChild*，他允许我们获取我们的*Nav*组件(下一步将会添加的)的直接引用。这很有必要因为我们不能在跟组件里面使用**NavController**，但是我们后续需要在这里改变**rootPage**所以我们直接获取**Nav**。

我们这里设置了一些函数，稍后会完成他们的定义。*openPage*函数将处理侧滑菜单（将在这里设置与*homePage*和*aboutPage*变量绑定）不同选择之间的切换，*logout*函数将用于登出当前应用。我们已经从Ionic Native中导入了Facebook插件，所以logout函数将与Facebook API交互。

同时，我们在装饰器里面用到了*templateUrl*连接到了一个**app.html**文件。大部分情况下我们都不需要担心根组件的独立模板文件，因为他太简单了（通常只是nav）。由于我们用来一个侧滑菜单，所以我们得自己去模板中定义这个菜单了，由于跟之前比有点多。所以我们现在来做这个。

> 在 **app** 文件夹内创建一个文件名为 **app.html**

```
<ion-menu [content]="content">
<ion-content>
<ion-list no-lines>
<button ion-item (click)="openPage(homePage)">
<ion-icon name="chatbubbles"></ion-icon> Chat
</button>
<button ion-item (click)="openPage(aboutPage)">
<ion-icon name="information-circle"></ion-icon> About
</button>
<button ion-item (click)="logout()">
<ion-icon name="power"></ion-icon> Logout
</button>
</ion-list>
</ion-content>
</ion-menu>
<ion-nav id="nav" #content [root]="rootPage"></ion-nav>
```

这里有些比较有趣的东西。着部分代码可能你看起来比较眼熟：

```
<ion-nav id="nav" #content [root]="rootPage"></ion-nav>
```

只有点点例外，也就是*#content*。 # 语法用于在模板定义本地变量，所以我们这里做的就是将指派给本地变量*content*。然后我们在我们的中使用这个变量来正确的设置他的内容：

```
<ion-menu [content]="content">
```

所以我们这里做的实际上就是告诉菜单他本身需要附加的内容是啥，在本例中也就是我们的整个应用。里面的其他内容只是菜单将要包含的内容，我们所作的知识创建一个按钮列表用作标签栏作用给用户点击。每个标签栏都会调用我们在根组件中定义的函数。
布局拼图的最后一篇是创建主页，我们先从模板入手。
> 修改 **src/pages/home/home.html**为如下：

```
<ion-header>
    <ion-navbar color="primary">
        <button ion-button icon-only menuToggle>
            <ion-icon name="menu"></ion-icon>
        </button>

        <ion-title>
            <img src = "assets/images/logo.png" class="logo" />
        </ion-title>
    </ion-navbar>
</ion-header>

<ion-content class="home" #chat id="chat">
    <ion-list no-lines>
        <ion-item>
            <ion-avatar item-left>
            <img src="">
            </ion-avatar>
            <h2>Name here</h2>
            <p>message here</p>
        </ion-item>
    </ion-list>
</ion-content>

<ion-footer>
    <ion-toolbar>
        <ion-input [(ngModel)]="chatMessage" type="text" placeholder="enter message..."
></ion-input>
        <button ion-button icon-only (click)="sendMessage()" style="position:absolute;
 right: 0; top: 0; margin: 0;">
            <ion-icon name="chatbubbles"></ion-icon>
        </button>
    </ion-toolbar>
</ion-footer>
```

我们在这个模板中重新用到了navbar，我们添加了一个按钮并给这个按钮添加了*menuToggle*属性这个属性将会添加一个按钮让用户可以切换菜单的开关（菜单可以通过划入和划出来打开和关闭，这就是为什么我们需要在登录页禁用他甚至我们都不添加一个菜单按钮）。

同时，请注意我们使用“#chat”定义了一个本地变量，也就是，这是因为我们稍有会通过*@ViewChild*来获取他的引用，我之前说过我们用作导航组件。

在内容区域里，我们创建了一个列表，目前其中只有一个（后续我们会用**ngFor*来展示所有信息），我们也用到了我们可以通过他给图片添加一些不错的样式。实际上我们将会在此处添加贴出信息的用户的名字，以及信息。

我们也用到了一个，他相当于一个缩减了所有导航魔法的navabar。这是一个我们可以添加标题，按钮或者其他东西的条，在本例中，我们将它放到屏幕底部，而不是顶部。这样我们可以保有一个信息输入区域即使在列表滚动的情况下他都死死的抱紧屏幕底部。

在toolbar里面我们加入了一个输入框，并通过*[(ngModel)]*将他与*charMessage*进行双向数据

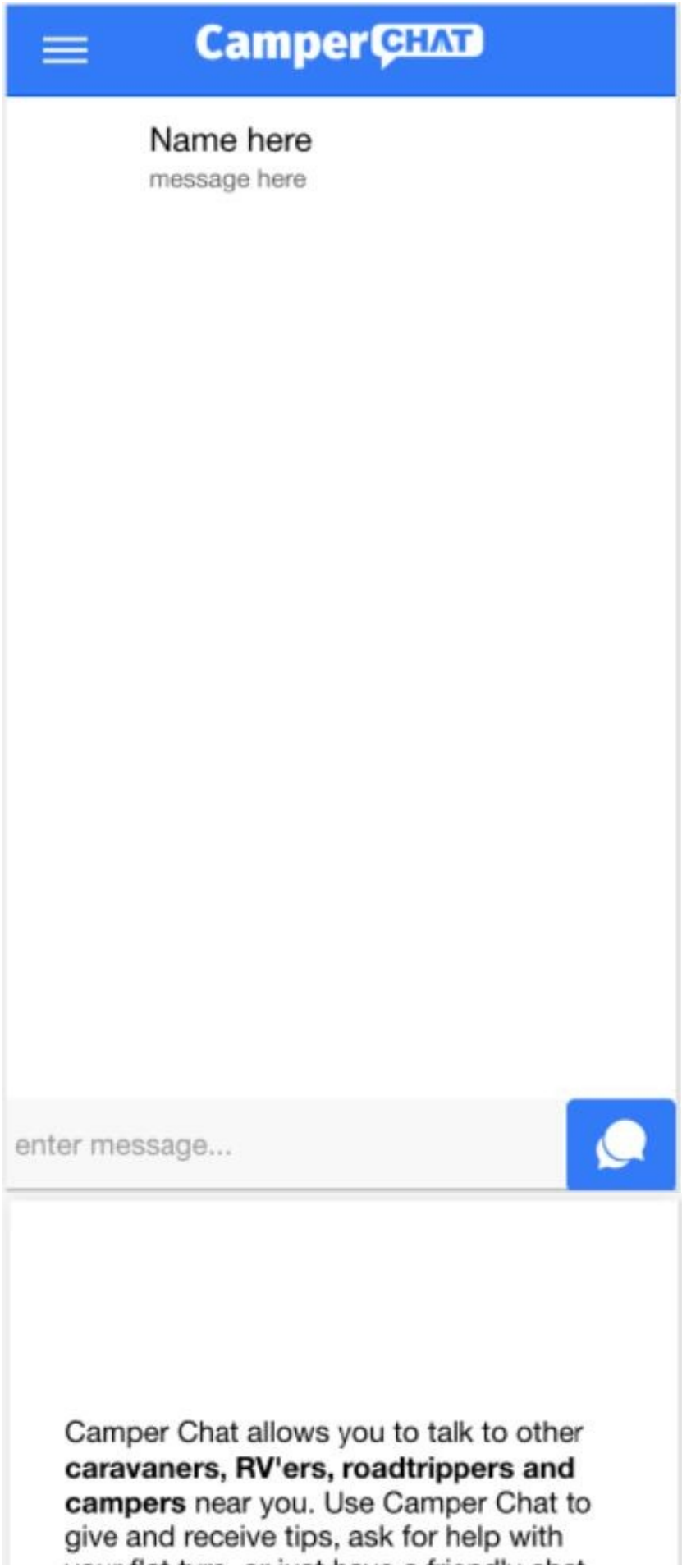
绑定。这就意味着当他的值变化的时候，**homt.ts**里面对应的*this.chatMessage*（稍后创建）也会改变（反之亦然）。然后我们就添加了一个按钮用于发送输入的信息，我们添加了一些内置样式让他浮在toolbar的右边。
现在来创建类定义，我们的**home.ts**文件。
> 修改 **src/page/home/home.ts** 为如下：

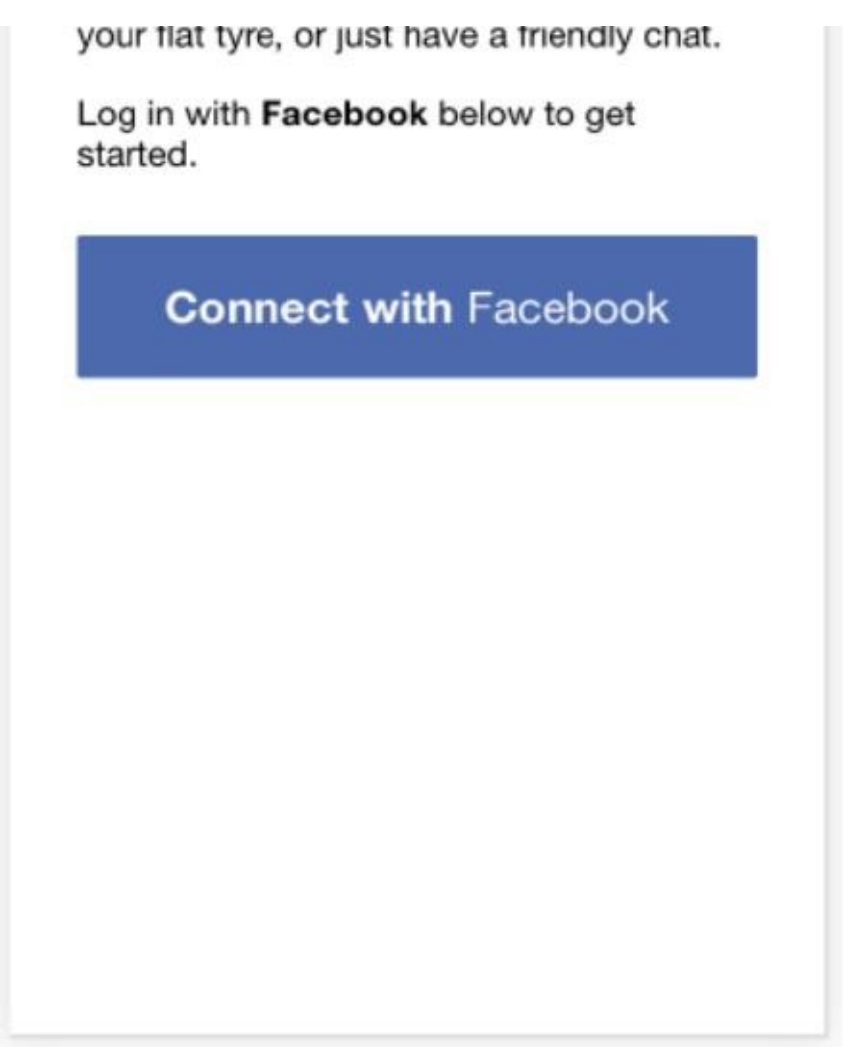
```
import { Component } from '@angular/core';
import { Data } from '../../providers/data';

@Component({
    selector: 'page-home',
    templateUrl: 'home.html'
})
export class HomePage {

    chatMessage: string = '';
    messages: any = [];

    constructor(public dataService: Data){

    }

    sendMessage(): void {

    }
}
```

这里蛮简单的，我们再次导入和注入了Data服务。我们设置好了模板内已经进行双向绑定的*this.chatMessage*和一个空的信息数组用来存储展示到屏幕的信息。
此时，运行应用的时候：

```
ionic serve
```

可以查看两个页面，大概是这样的：





不出意料，看起来还是蛮丑的，但是我们需要的基础布局已经摆在那里了。我们有一个带有登录选项的登录页，将会把我们引导至一个带有侧滑菜单的主页。

下节课我们将开始整合Facebook API这样我们可以让用户真正的登录（同时他可以给我们提供一些会在应用中用到的数据，像是用户名啦，用户头像啦）。

第四课：使用Facebook做授权验证

本节课中我们将整合Ionic Native中的Facebook插件，这样用户就可以在应用中获得认证，同时我们也可以做其他的事情例如获得用户信息。使用Facebook可以做大把的事情，我不会一一赘述，你还是看文档来的爽快些。

使用类似Facebook API类似的社交认证给我们带来了大量的好处。他帮助我们节省了时间去创建自己的认证后端服务（这里面包括了创建数据库，存储用户安全证书，操作密码重设逻辑等等），他帮助用户节省了创建新帐户的精力（他们只要一键就可以登录了），他提供了大量好用的数据和整合。这也不是说社交登录同源是最佳的认证方法，但是他的优点也很明显。

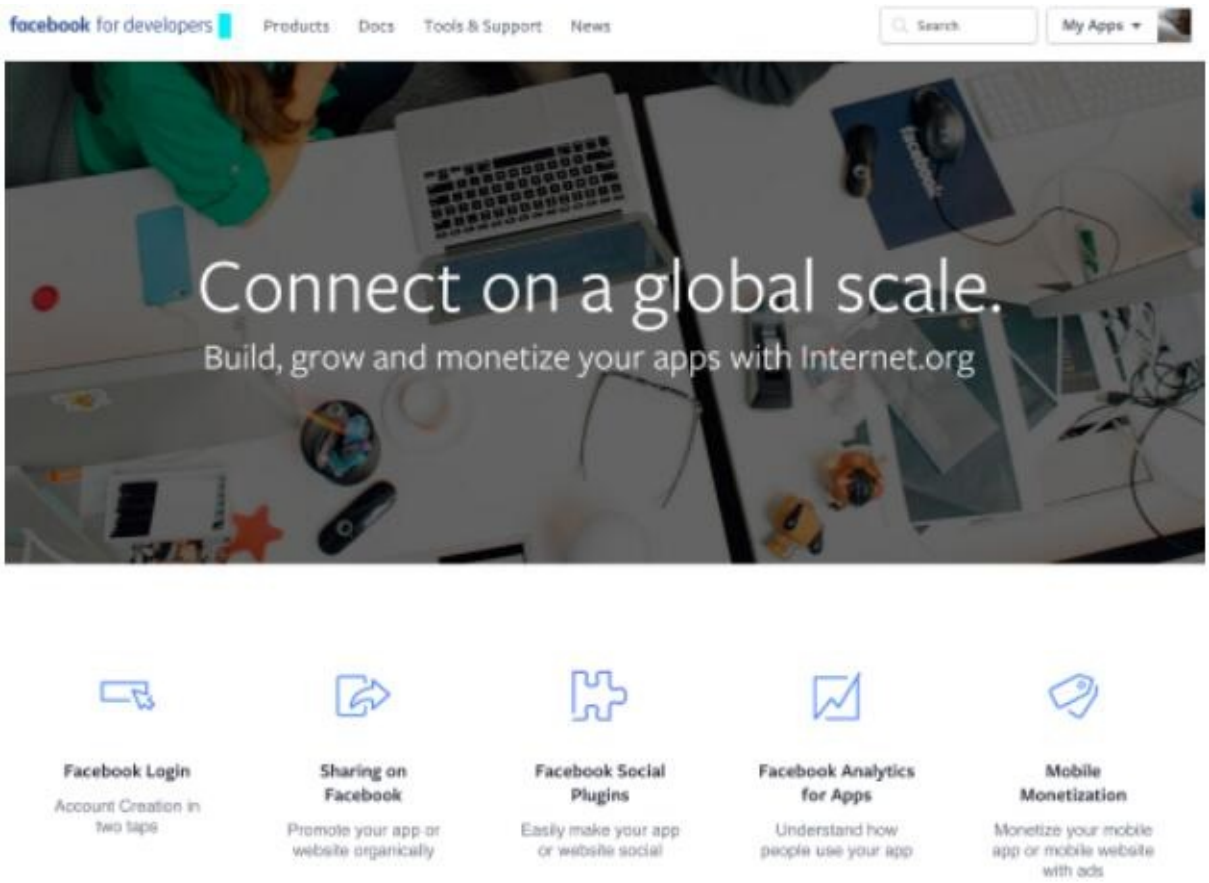
我们来研究一下如何设置Facebook插件然后将他整合到我们的应用中来。

重要：Facebook连接插件只会在真机上运行的时候在有效，如果你在桌面浏览器中运行的时候只会受到错误。用*ionic run android*或者*ionic run ios*代替*ionic serve*。如果你不确定怎么样才能让应用在你的设备上运行的话，先看一下本书的测试与调试部分。

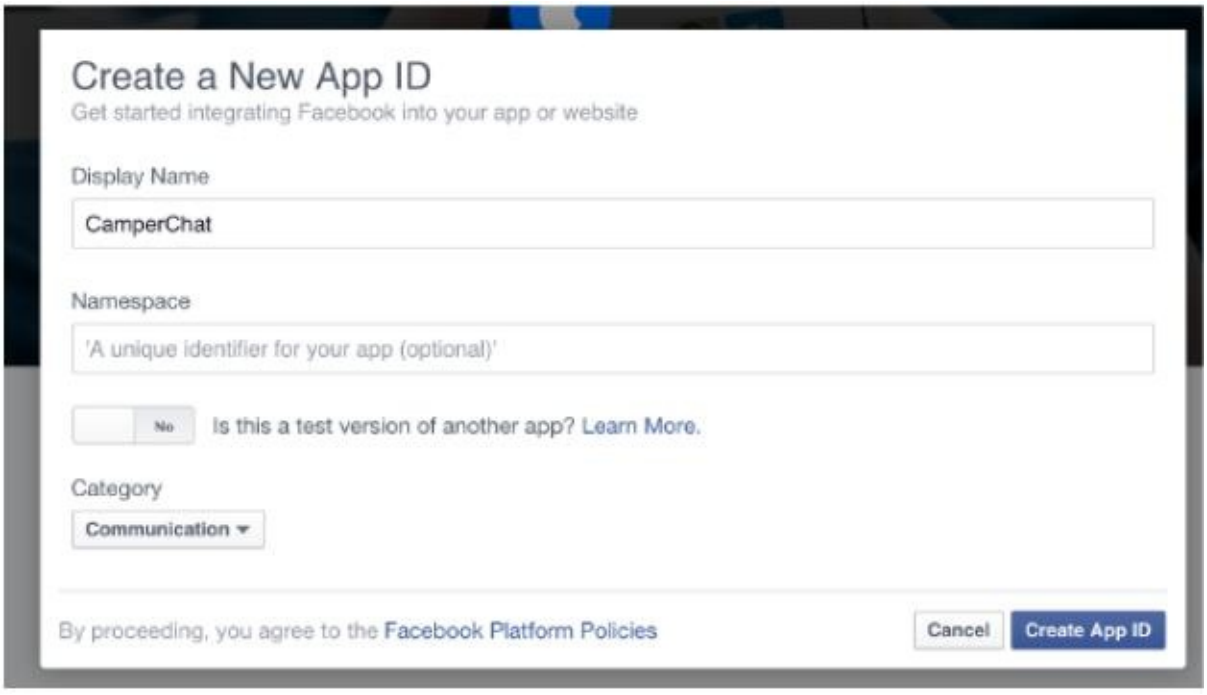
设置Facebook应用

之前课程讲过，这个插件跟普通插件安装不一样。我们在安装的时候需要给插件提供一个**App Name**和一个**App ID**。所以我们需要在Facebook开发者平台创建一个应用，我们先来进行这一步。

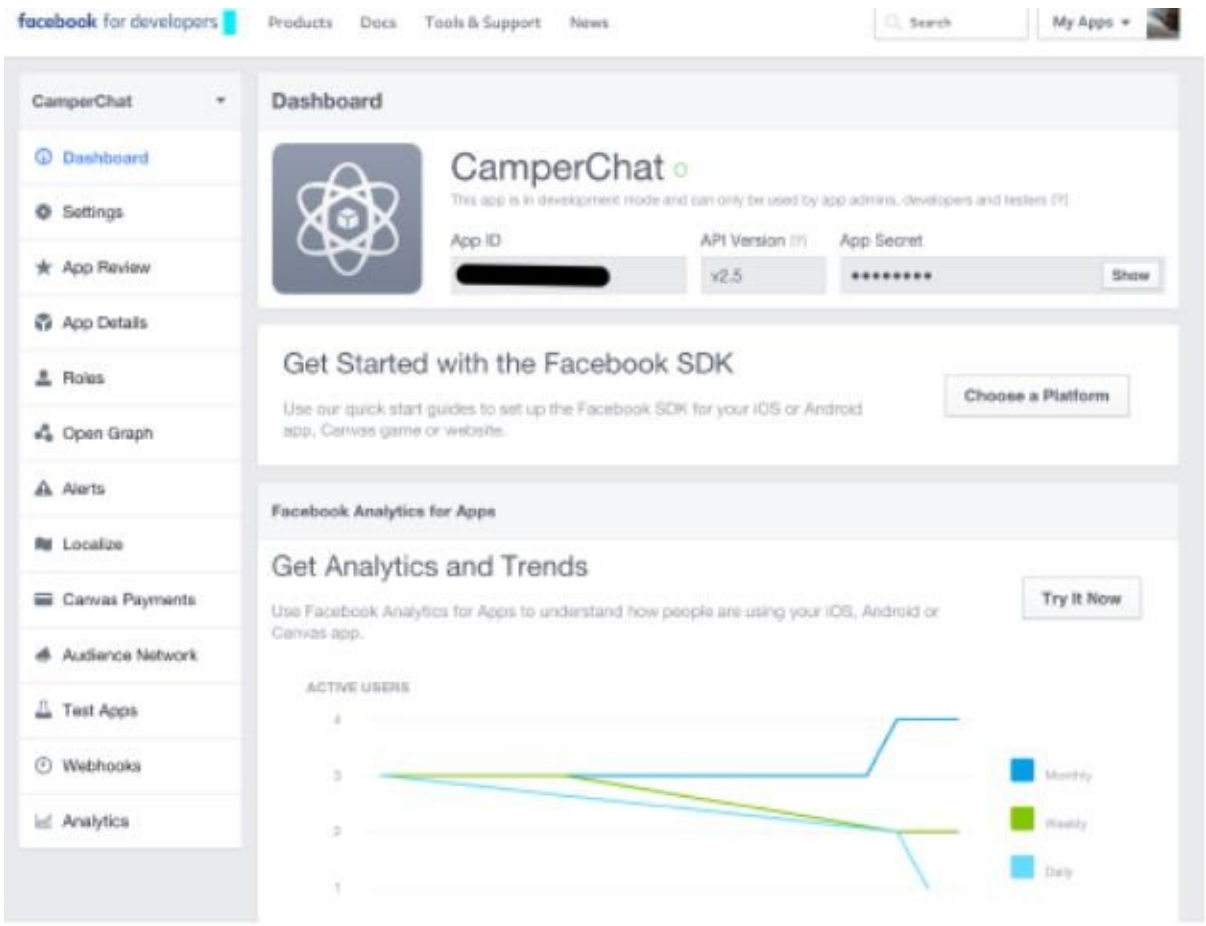
首先你需要去developers.facebook.com如果没有帐号的话麻烦先注册一个。如果做完这一步的话，你应该可以看到如下：



点击**My Apps**然后**Add a New App**。选择**iOS**平台，然后在下一页上点击右上角的**Skip and Create App ID**按钮。填好**Display Name**（你的应用的名字，我们现在假定是CamperChat）并选择一个分类。然后点击**Create App ID**。



现在你应该来到这样的一个屏幕。



你应该可以看到一个域叫做**App ID**，把他记在本子上因为我们马上就要用到他了。我们还需要一些设置。点击左边的**Settings**标签，在此我们将要添加使用平台，也就是iOS和Android。

我先现在可以添加iOS平台，但是Android平台在保存之前需要一个“key hash”。所以，我们现在不理他了，但是后面一定要回来加好因为没有他Android平台将无法运行。我们将在后面的构建章节学习如何为Android创建一个keystore文件和一个key hash，所以如果你现在就像完成这一步的话，可以现在就去到 Signing Android Applications部分。

点击**Add Platform**按钮然后选择**iOS**。你需要添加一个**Bundle ID**大概是这样的格式*com.*你的名字字母.你的项目或者*com.*你的公司.你的项目或者其他类似的东西。（译者：注意，一定要用字母，不要让我后悔把包名翻译了一下）同时你也需要确保将**Single Sign On**为‘Yes’。

同时需要确保你提供的Bundle ID和你的Ionic项目中的**config.xml**中的一致，所以确认一下你的**config.xml**下面这一行是否跟你的Facebook Apps Bundle ID匹配：

```
<widget id="com.yourname.yourproject" version="0.0.1"
    xmlns="http://www.w3.org/ns/widgets"
    xmlns:cdv="http://cordova.apache.org/ns/1.0">
```

在后面将你的Facebook app添加Android平台的时候也要提供相同的Bundle ID（他应该是在Google Play Package Name下面添加的）。

这是目前为止能做的，**App Name**和**App ID**在手，我们就可以给应用安装插件了。

重要：当你的应用准备上线的时候，记得将Facebook应用切换出开发模式，也就是去App Review标签页切换“Make Your App Public？”为“Yes”。

安装Facebook连接插件

在应用中安装此插件的话需要在项目目录内运行如下命令：

```
ionic plugin add cordova-plugin-facebook4 --save --variable APP_ID="123456789" --varia
ble APP_NAME="myApplication"
```

记得将APP_ID和APP_NAME替换成你自己刚创建的的Facebook应用的。
重要：目前在Andoid上这个插件还有一个构建问题。根据你使用的版本这个可能也可能不会影响到你，但是如果构建Android失败的话，你可能需要去更改*platforms/android/project.properties*这一行：

```
cordova.system.library.1=com.facebook.android:facebook-android-sdk:4.4.0
```

设置认证

我们现在设置好了我们所需的东西，我们现在可以进行一点点编码工作了。整合Ionic Native的Facebook API其实非常简单，但是在这之前我们先创建我们的数据服务这样我们可以临时存储一些Facebook的相关值。数据服务主要是用于处理存储和获取我们的信息数据，但是由于我们需要存储一些来自Facebook的数据，所以很容易在这里确定好。
> 修改 **src/providers/data.ts** 为如下：

```
import { Injectable } from '@angular/core';

@Injectable()
export class Data {
    fbid: number;
    username: string;
    picture: string;

    constructor() {

    }
}
```

现在，我们要对登录页进行一些修改，以让用户在点击登录按钮的时候通过Facebook验证，而不是直接去到主页。
> 修改 **src/pages/login/login.ts** 的 **login** 函数如下：

```
login(): void {
    Facebook.login(['public_profile']).then((response) => {
        this.getProfile();
    }, (err) => {
        let alert = this.alertCtrl.create({
            title: 'Oops!',
            subTitle: 'Something went wrong, please try again later.',
            buttons: ['Ok']
        });

        alert.present();
    });
}
```

由于我们已经从Ionic Native导入过**Facebook**，我们现在可以访问**Facebook**对象了。我们调用*login*方法并传入我们请求的权限数组。我们只需要基本用户信息，所以我们只请求了用户的“public profile”权限，其他一些可以请求的权限例如：

- 用户朋友
- 用户兴趣
- 用户简介
- 用户位置
- 用户打标签地方

还有其他一大堆，想要知道的话，点我吧。*login*函数返回一个promise，所以我们给响应设置了一个处理器，如果成功的话我们触发之前设置的*getProfile*函数，这个函数将启动主页。如果不成功的话，我们创建一个警告框并展示给用户。
我们现在知道用户成功通过Facebook认证，他们也连接到我们的应用了，但是我们还是需要搞清楚一些细节。为了让他们能够用上我们的应用，我们需要他们的名字和照片，为了获取这些信息我们需要需改*getProfile*函数。
> 修改 **src/pages/login/login.ts** 的 **getProfile** 函数如下：

```
getProfile(): void {

    Facebook.api('/me?fields=id,name,picture', ['public_profile']).then((response) =>
{
        console.log(response);
        this.dataService.fbid = response.id;
        this.dataService.username = response.name;
        this.dataService.picture = response.picture.data.url;
        this.menu.enable(true);
        this.nav.setRoot(HomePage);
    },
    (err) => {
        console.log(err);
        let alert = this.alertCtrl.create({
        title: 'Oops!',
        subTitle: 'Something went wrong, please try again later.',
        buttons: ['Ok']
        });
        alert.present();
    });
}
```

现在我们想Facebook插件提供的api发起了调用。他允许我们与 Facebook Graph API 交互这就是我们能用Facebook做的所有事情了 -- 你可以做一些比较复杂的东西了但是我们只是用它来获取用户id，全名和展示图片。
第一个需要提供给api调用的参数是需要访问的终端，当前对我们来说就是*/me*因为我们想要摘掉当前登录用户的数据，我们添加了一个查询字符串包含了所有我们想要返回的数据域。我们也提供了一个权限数组然后等待响应。
如果响应成功的话我们使用我们的dataService存储得到的数据，然后我们启用侧滑菜单并转到主页。如果返回不成功的话，我们就要再次给用户展示一个警告框并把停留在登录页。
另一个需要用到的功能是Facebook API的登出方法。当用户登出我们的应用的时候我们也需要Facebook终止本次会话（session）（他不会登出facebook，只是再次使用的时候需要通过Facebook再次重新认证）。所以我们修养修改我们的*logout*函数。
> 修改 **src/app/app.module.ts**的**logout**方法如下：

```
logout(): void {
    this.menu.close();
    this.menu.enable(false);
    this.nav.setRoot(LoginPage);
    this.dataService.fbid = null;
    this.dataService.username = null;
    this.dataService.picture = null;
    Facebook.logout();
}
```

首先我们处理我们这边的事情。我们关闭了菜单并禁用了他，然后切换回登录页。我们重置了所有存放在我们数据服务是的Facebook数据，然后我们调用了Facebook API的*logout*方法。现在用户应该在咱们的应用中回到了登录页并且是未认证状态。
由于用户在认证的时候会有一点点的等待时间（就像我们获取用户资料的时候一样），我们想要给用户展示正在发生着一些事情，而不是应用卡死了。我们会用Ionic的**LoadingController**服务来完成这个。他允许我们在一个时期内停止和应用交互。
> 修改 **src/pages/login/login.ts** 为如下：

```
import { Component } from '@angular/core';
import { Platform, NavController, MenuController, AlertController, LoadingController }
 from 'ionic-angular';
import { Facebook } from 'ionic-native';
import { HomePage } from '../home/home';
import { Data } from '../../providers/data';

@Component({
    selector: 'page-login',
    templateUrl: 'login.html'
})
export class LoginPage {

    loading: any;

    constructor(public nav: NavController, public platform: Platform, public menu: Men
uController, public dataService: Data, public alertCtrl:AlertController, public loadin
gCtrl: LoadingController) {
        this.loading = this.loadingCtrl.create({
            content: 'Authenticating...'
        });
        this.menu.enable(false);
    }

    login(): void {
        this.loading.present();

        Facebook.login(['public_profile']).then((response) => {
            this.getProfile();
        }, (err) => {
            let alert = this.alertCtrl.create({
                title: 'Oops!',
                subTitle: 'Something went wrong, please try again later.',
                buttons: ['Ok']
            });

            this.loading.dismiss();
            alert.present();

        });
    }
```

```
    getProfile(): void {
        Facebook.api('/me?fields=id,name,picture', ['public_profile']).then((response)
 => {
            console.log(response);
            this.dataService.fbid = response.id;
            this.dataService.username = response.name;
            this.dataService.picture = response.picture.data.url;
            this.menu.enable(true);
            this.loading.dismiss();
            this.nav.setRoot(HomePage);
        },
        (err) => {
            console.log(err);
            let alert = this.alertCtrl.create({
                title: 'Oops!',
                subTitle: 'Something went wrong, please try again later.',
                buttons: ['Ok']
            });

            this.loading.dismiss();
            alert.present();
        });
    }
}
```

注意在这里我们导入了**LoadingController**服务，我们在构造器中设置了一个“Authenticating...”信息。然后我们在调用登录方法的时候呈现了这个加载信息（就像Alert或者Modal一样）。然后，在响应的时候不管返回成功失败我们都关闭了加载中的信息。

这就是应用整合Facebook的全部了，但是可以通过Facebook API能做的远远不止这些。建议你看一下插件文档和Facebook Graph API了解更多。

下节课中，我们来研究怎么样向屏幕上发送信息。

第五课：创建信息和导航

明显这是我们应用中最重要的功能之一，但是实现起来蛮简单的。最难的部分是整合PouchDB和Cloudant，我们稍后实现，目前我们只要给登录的用户提供想屏幕上添加信息的能力就可以了。
我们也会利用导航来绑定一些东西 -- 我们有一个带有选项的菜单但是啥事都没做，和一个什么都没有的about页。

添加信息

在咱们的home也类定义中，我们已经设置了输入框和*this.chatMessage*变量的绑定，以及一些按钮来触发*sendMessage*函数，所以启用创建信息的能力只需要去实现*sendMessage*函数即可。我们还要修改模板来循环展示所有可以信息。
> 修改 **src/pages/home/home.ts** 的**sendMessage**函数为如下：

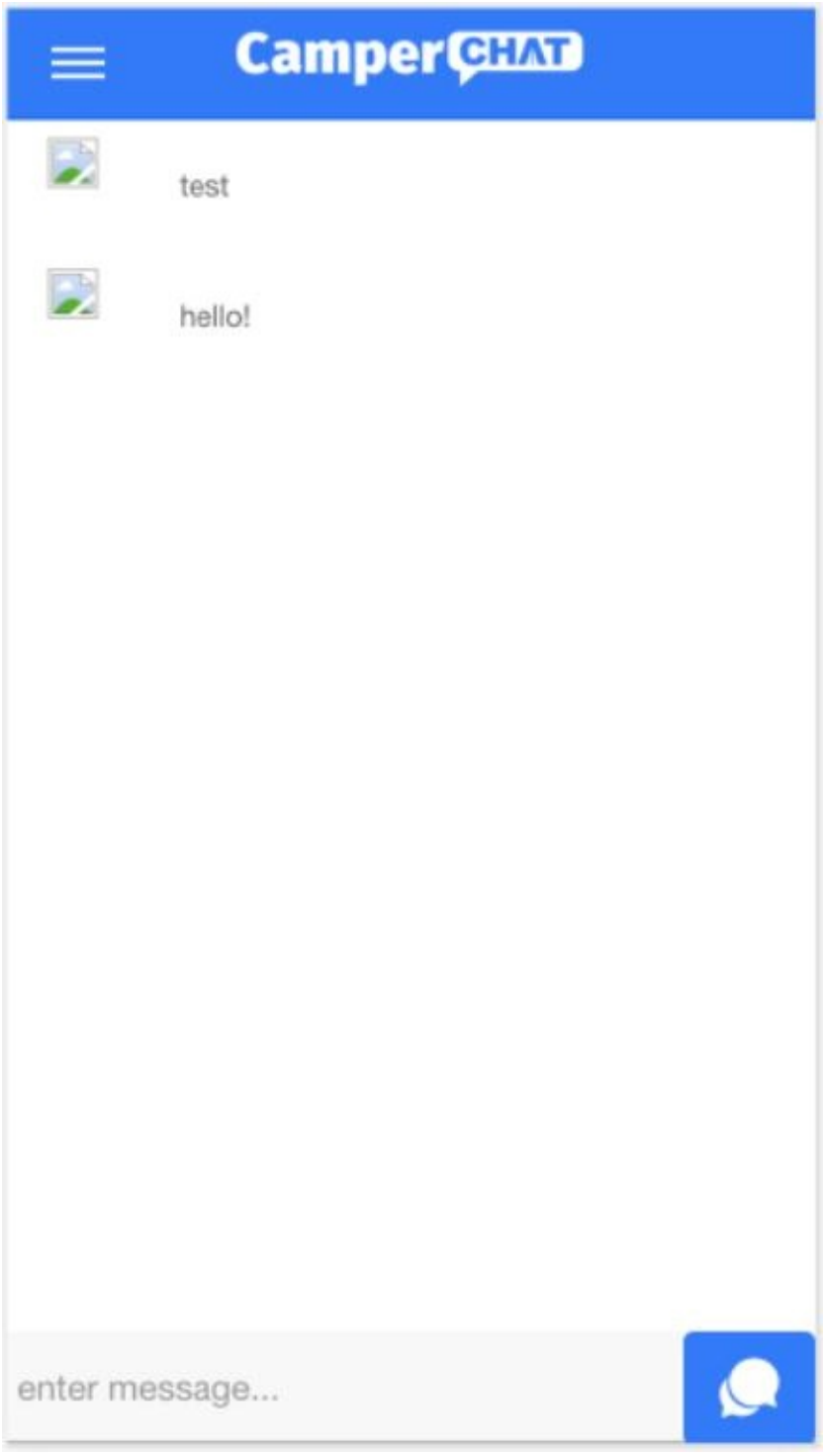
```
sendMessage(): void {
    let message = {
        '_id': new Date().toJSON(),
        'fbid': this.dataService.fbid,
        'username': this.dataService.username,
        'picture': this.dataService.picture,
        'message': this.chatMessage
    };
    this.messages.push(message);
    this.chatMessage = '';
}
```

在这里我们使用一些属性创建了一个*message*对象。我们加入了刚刚从Facebook得到的所有信息，以及用户输入的信息。我们也添加了一个“id”字段给下节课整合的PouchDB和Cloudant使用，目前请无视。
完成信息的创建之后我们将他压入信息数组然后重设聊天信息输入框（这样用户可以接着用来输入信息了）。我们下节课再对他进行修改让这些信息可以发送到远程后端，而不是直接压入到信息数组。
现在我们有了将一些数据压入信息数组的能力，我们修改模板来展示他们。
> 修改 **src/pages/home/home.html** 的列表如下：

```
<ion-list no-lines>
    <ion-item *ngFor="let message of messages">
        <ion-avatar item-left>
            <img [src]="message.picture">
        </ion-avatar>

        <div>
            <h2>{{message.username}}</h2>
            <p>{{message.message}}</p>
        </div>
    </ion-item>
</ion-list>
```

现在我们循环所有的信息并将它们显示到他们自己的中。记住'ngFor'是创建嵌入模板的快捷方式，'ngFor'将为**messages**里面的每个**message**创建一个嵌入模板并使用指定的信息进行渲染。我们可以通过message提供的数据展示用户头像，用户名还是信息本身。

如果你现在运行应用的话，你应该可以添加一些测试信息到应用并看到他们显示出来（记住，必须在真机上测试因为Facebook登录在ionic serve中不会正常工作）：



制作About页

我们设置好了策划菜单，但是实际上现在能用的只有'Logout'按钮。我们还需要制作about也并连接过去，还有'Chat'按钮和主页的连接。我们先来实现about页。

> 修改 **src/pages/about/about.html** 为如下：

```
<ion-header>
<ion-navbar color="primary">
<button ion-button icon-only menuToggle>
<ion-icon name="menu"></ion-icon>
</button>
<ion-title>
<img src = "assets/images/logo.png" class="logo" />
</ion-title>
</ion-navbar>
</ion-header>
<ion-content padding>
<p>Camper Chat allows you to talk to other <strong>caravaners, RV'ers,
roadtrippers and campers</strong> near you. Use Camper Chat to give
and receive tips, ask for help with your flat tyre, or just have a
friendly chat.</p>
</ion-content>
```

很简单的页面，没啥好看的。和home页一样我们实现了*menuToggle*按钮，然后添加了一些描述性的内容。
现在我们来设置菜单上点击处理器了。如果现在看一眼**app.html**的话发现这些按钮都是调用的*openPage*函数：

```
<button ion-item (click)="openPage(homePage)">
    <ion-icon name="chatbubbles"> Chat</ion-icon>
</button>
<button ion-item (click)="openPage(aboutPage)">
    <ion-icon name="information-circle"> About</ion-icon>
</button>
```

只是一个传入'homePage'另一个传入'aboutPage'。我们现在设置*openPage*函数。
> 修改 **src/app/app.component.ts** 的 **openPage** 函数如下：

```
openPage(page): void {
    this.menu.close();
    this.nav.setRoot(page);
}
```

首先我们关闭了菜单，因为用户进行了选择（如果菜单还是停止那里的话挺烦人的）然后通过我们的nav组件更改了根页面。我们将它设置为传入的页面参数，因为我们已经设置好了*homePage*和*aboutPage*的引用：

```
homePage: any = HomePage;
aboutPage: any = AboutPage;
```

当调用此函数的时候会转到对应的页面上去。现在用户可以来回切换about页和chat页了，然后他们也可以登出返回到登录页。
我们现在设置好了所有的基本功能，还有一件要做的大师赛整合PouchDB和Cloudant，下节课的内容，然后整理整理让他变漂亮点就可以了。

第六课：本地和远程PouchDB和Cloudant后台

虽然我们完成了应用的大部分功能，但是本节课将是最大最难的那个。我们将使用PouchDB来存储信息而不是直接将他扔到信息数组里面去。

PouchDB是一个浏览器内的NoSQL数据库，灵感来自CouchDB项目。他最大的功能是允许存储离线数据，当应用再次上线之后会自动从远程数据库同步数据。和使用Ionic提供的SqlStorage一样，使用PouchDB保证你存储的本地数据不会被随机擦除。

普及NoSQL不是本书的目的，但是为了给你一点基础，NoSQL数据库通常以类JSON的键值对风格格式来存储数据，不同的样式要不同的去思考。所以，所以当你来到NoSQL的时候，需要放弃你可能已有大量的的SQL固有观念（假设你之前用过）。我们存储的数据非常简单，所以我们不需要担心如何合理的使用NoSQL数据库。

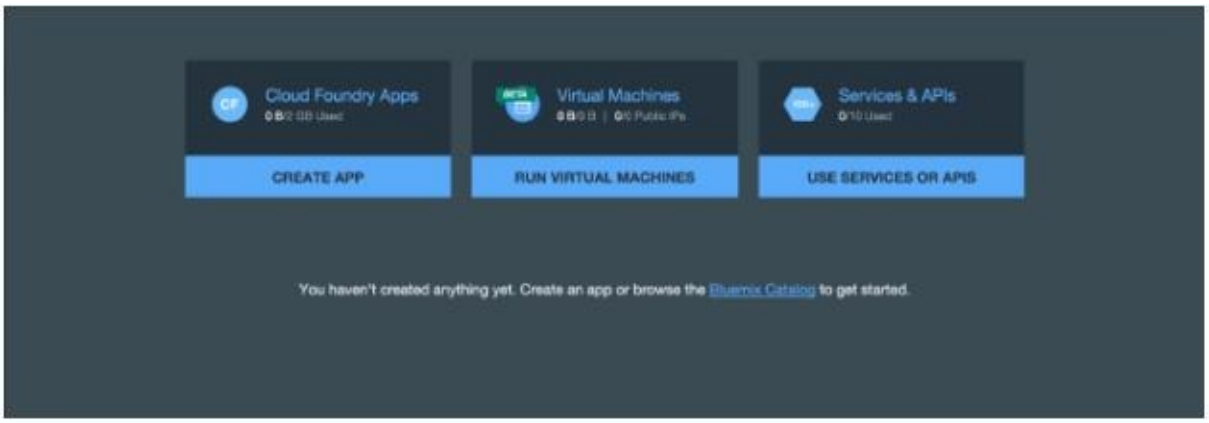
所以PouchDB会负责本地数据的存储，但是我们还要用到Cloudant来创建一个远程数据库。我们将同步本地的PouchDB数据库和远程的Cloudant数据库，这样无论何时应用上线的时候都可以从Cloudant获取最新数据，任何本地新数据都将推送到Cloudant。幸运的是，这两个工具都被设计为相互协作且设置好了远程同步，这在平常事一个非常复杂的任务，现在却变得非常简单了。

Cloudant是一个DBaas（Database as a Service数据库即是服务），所以我们只要创建一个帐号（低端使用免费）来使用，当然也要设置数据库。使用类似Cloudant这样的的东西非常简单，且扩展非常简单（因为所有后端架构都帮你处理好了），但是如果你喜欢的话，你也可以轻松的使用类似CouchDB或者Couchbase安装到你的服务端来和PouchDB同步。这个我不会深入，因为差别不大。

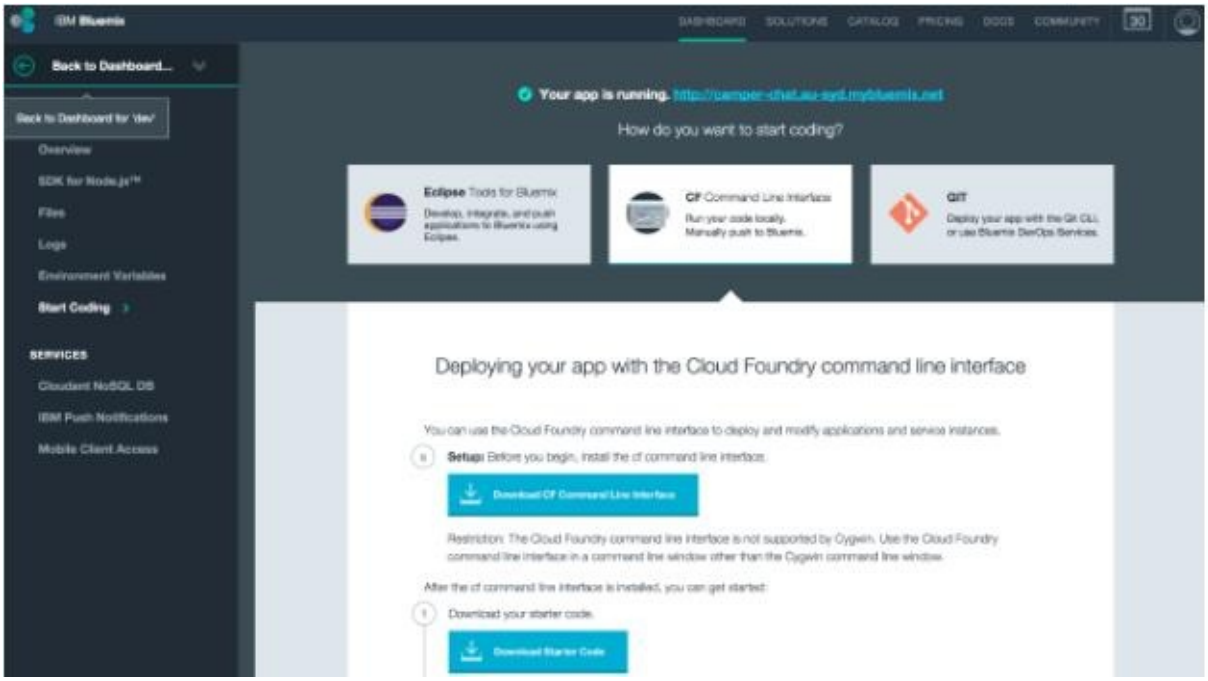
创建Cloudant Database

在进入编码之前，我们要在IBM Bluemix的Cloudant上设置我们的后端。Bluemix让我们可以访问IBM的Open Cloud Architecture，他可以用来创建，部署和管理基于云的应用。他提供大量可用的服务，其中一个就是Cloudant。

首先你的创建一个IBM Bluemix帐号。当你创建好帐号登录进去之后，可以看到这样的界面：

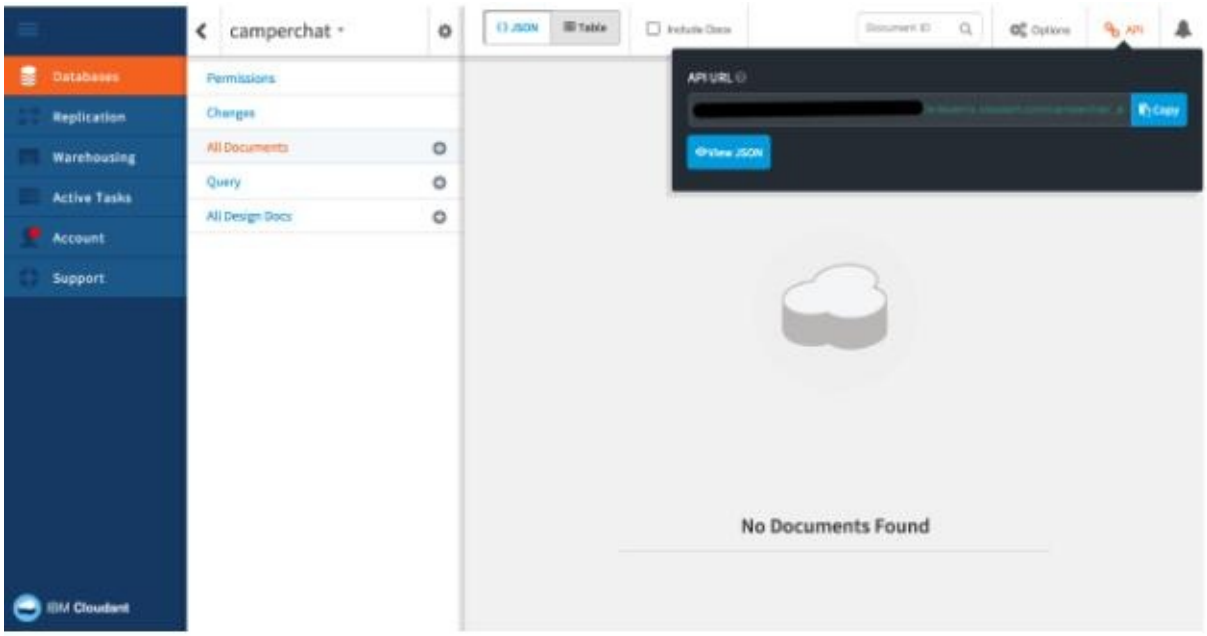


选择Create App选项，选择Mobile，给你的应用取个名字然后点击Finish。之后就会看到这样的画面：



从左边的Service菜单选择Cloudant NoSQL DB，然后在Cloudant Dashboard上点击View your Data。你现在应该来带了Cloudant的仪表盘了。点击右上的Create Database选项创建一个新的数据库然后命名为**camperchat**或者其他你喜欢的。

现在，选择刚才新建的数据库查看具体细节，应该可以看到如下画面：



点击右上的API连接就得到了你的Cloudant Database URL了，这个我们要稍后提供给PouchDB的，所以最好拿个本子记下来吧。同时需要去Permissions部分生产一个API密钥（确保给他提供Write和Replicate权限） -- 他可以让PouchDB访问你的数据库，使用API密钥比用你的用户名和密码好一些。请记住密码因为一旦离开屏幕你就再也找不到了。

这里我们还有一件事情要做。我们需要激活CORS(Cross Origin Resource Sharing)这样我们就可以从我们的应用中向数据库发起请求了。来到左边菜单的Account，选择CORS然后选择All Domains(*) 选项。

这样就在Cloudant上设置好了，可以跟PouchDB一起使用了。

整合PouchDB

之前我们创建好了Data服务，也只是存放了一点Facebook API返回的值。现在我们要扩展Data服务用来操作PouchDB的数据存储和获取。

> 修改 **src/providers/data.ts** 为如下：

```
import { Injectable } from '@angular/core';
import PouchDB from 'pouchdb';

@Injectable()
export class Data {
    fbid: number;
    username: string;
    picture: string;
    db: any;
    data: any;
    cloudantUsername: string;
    cloudantPassword: string;
    remote: string;

    constructor(){
        this.db = new PouchDB('camperchat');
        this.cloudantUsername = 'YourAPIUsernameHere';
        this.cloudantPassword = 'YourAPIPasswordHere';
        this.remote = 'https://YOUR-URL-HERE-bluemix.cloudant.com/camperchat';
        //Set up PouchDB
        let options = {
            live: true,
            retry: true,
            continuous: true,
            auth: {
                username: this.cloudantUsername,
                password: this.cloudantPassword
            }
        };
        this.db.sync(this.remote, options);
    }

    addDocument(message){
    }

    getDocuments(){
    }

    handleChange(change){
    }
}
```

我们已经通过**npm**安装好了PouchDB，想要在这个服务里使用的话我们得先导入：

```
import PouchDB from 'pouchdb';
```

在构造器中，我们处理了PouchDB的设置和远程Cloudant数据库的同步。首先我们我们新建了一个PouchDB，或者获取一个已存在的引用：

```
this.db = new PouchDB('camperchat');
```

然后我们定义了一些用于连接到Cloudant数据库的变量：

```
this.cloudantUsername = 'YourAPIUsernameHere';
this.cloudantPassword = 'YourAPIPasswordHere';
this.remote = 'https://YOUR-URL-HERE-bluemix.cloudant.com/camperchat';
```

请记得要用你自己的Cloudant仪表盘上的参数替换上面这些值。接着我们创建了一个options对象来配置到Cloudant数据库的连接，然后我们调用了*sync*方法：

```
this.db.sync(this.remote, options);
```

这样将会设置从PouchDB数据库复制到Cloudant数据库，同时也会设置从Cloudant数据库到PouchDB数据库的复制。现在，如果我们给PouchDB添加一些数据的时候将会自动反射到远程Cloudant数据库，如果我们在远程Cloudant修改或者添加一些数据的时候，将会自动反射到我们的本地数据库。
在那之后我们创建了三个空函数，我们现在就来实现。

addDocument

> 修改 **addDocument** 为如下：

```
addDocument(message) {
    this.db.put(message);
}
```

我们只需要简单的调用数据库的*put*就可以向PouchDB添加文档（NoSQL条款里一个数据对象叫做一个“文档”，所以可以将文档想象成一小片数据，而不是一个Word文档）。这样我们可以向这个函数传入任何数据以存储到数据库。

getDocument

> 修改 **getDocument** 为如下：

```
getDocuments(){
    return new Promise(resolve => {
        this.db.allDocs({
            include_docs: true,
            limit: 30,
            descending: true
        }).then((result) => {
            this.data = [];

            let docs = result.rows.map((row) => {
                this.data.push(row.doc);
            });

            this.data.reverse();
            resolve(this.data);

            this.db.changes({live: true, since: 'now', include_docs:true}).on('change'
, (change) => {
                this.handleChange(change);
            });
        }).catch((error) => {
            console.log(error);
        });
    });
}
```

这个函数用户获取数据库中的所有文档（记住，文档只是一个数据对象）。这是一个一部操作，所以我们将他包装到一个promise里，然后在数据返回的时候在resolve。他允许我们在应用内其他地方使用*getDocuments().then()*语法。通过调用*allDocs*可以获取所有文档，这里我们也提供了一些选项：

```
include_docs: true,
limit: 30,
descending: true
```

这里的*include_docs*看起来有点迷糊，但是我们还是要指定这个这样文档内的所有数据都将被返回（信息，图片等）。如果没有提供这个参数的话只会返回文档的id。我们也设置了一个30的显示，和一个降序，这样一来只返回最新的30个文档（聊天信息）。
我们将这些结果传入到我们的处理器，每个返回的行我们都会压入到*this.data*数字。我们想让这些信息倒序这样的话最新的消息将会显示在最下面。之后我们**resolve**我们创建的promise然后回传数据，然后设置*changes*监听器。
*changes*监听器在每次侦测到数据库变更的时候（例如当其他用户添加了一个聊天信息的时候）都将调用*this.handleChange*函数，change本身也会被传入到函数里。我们现在来定义。

handleChange

> 修改 **handleChange** 为如下：

```
handleChange(change) {
    let changedDoc = null;
    let changedIndex = null;
    this.data.forEach((doc, index) => {
        if(doc._id === change.id){
            changedDoc = doc;
            changedIndex = index;
        }
    });

    //A document was deleted
    if(change.deleted){
        this.data.splice(changedIndex, 1);
    }
    else {
        //A document was updated
        if(changedDoc){
            this.data[changedIndex] = change.doc;
        }
        //A document was added
        else {
            this.data.push(change.doc);
        }
    }
}
```

我们将传入的change反射到我们的*this.data*数组，但是有点技巧。传入进来的change对象可能是一个文档更新了，新建了一个文档，或者删除了一个文档。
检查文档删除非常简单只要检查他是否有**deleted**属性就可以了。但是辨别他是否是更新，我们需要检查我们是否有了相同id的文档（如果找到了的话就更新他），如果没有的话我们就知道是新增文档了（我们需要将他添加到数组）。
现在我们完全设置好了Data服务，但是如果要使用他的话还有些事情要做。我们需要使用provider来存储新数据，而不是像现在这样直接将他添加到messages数组，同时在应用打开的时候我们需要加载最新的信息。我们现在就来完成这些。
> 修改 **src/page/home/home.ts** 如下：

```
import { Component, ViewChild } from '@angular/core';
import { Data } from '../../providers/data';

@Component({
    selector: 'page-home',
    templateUrl: 'home.html'
})
export class HomePage {

    @ViewChild('chat') chat: any;
    chatMessage: string = '';
    messages: any = [];

    constructor(public dataService: Data){
        this.dataService.getDocuments().then((data) => {
            this.messages = data;
            this.chat.scrollToBottom();
        });
    }

    sendMessage(): void {
        let message = {
            '_id': new Date().toJSON(),
            'fbid': this.dataService.fbid,
            'username': this.dataService.username,
            'picture': this.dataService.picture,
            'message': this.chatMessage
        };

        this.dataService.addDocument(message);
        this.chatMessage = '';
    }
}
```

现在我们在构造器中调用了*getDocument()*函数来加载信息数据，一旦完成之后我们通过*@ViewChild*获取滚动内容的引用，将他滚动到底部。
最后，在*sendMessage*函数中，唯一的变更是我们调用了*addDocument*函数而不是直接把信息压入到*messages*数组。
终于完成了！【译者：我尿也憋坏了】现在聊天应用的全部功能都开发完了。他能够正常工作了，但是也丑爆了。下节课我们让他变漂亮点甚至给他加点动画的啥的！

第七课：自定义样式与动画

这节课中我们将稍稍改动一下模板添加一些类，我们还是创建一些自定义的样式，同时也将覆盖一些应用级的SASS变量。如果之前完成过其他原因，那么你基本知道本节课的实际内容不多，没有那么难，唯一不同的是我们将添加一些自定义动画。
动画，使用得当的话，可以让你的应用看起来，感觉起来质量更高。当使用不当的时候，适得其反，甚至引发效率问题。

基本样式

我们先添加全局基本样式让他稍微漂亮些。首先，我们将要修改**variables**文件进行大范围的修改。
> 修改 **src/theme/variables.scss**文件如下：

```
$colors: (
    primary: #5b91da,
    secondary: #32db64,
    danger: #f53d3d,
    light: #f4f4f4,
    dark: #222,
    favorite: #69BB7B
);

$list-ios-border-color: #fff;
$list-md-border-color: #fff;

$button-ios-border-radius: 0px;
```

我们这里没有做太多，只是定义了一些我们需要用到的颜色，为Android和iOS设置了默认的边沿色，同时对iOS设置了按钮的圆角为0 px。（Android上按钮本来就是方形的）
同时我们需要去核心样式文件中定义一些应用级的样式。
> 将一下样式添加到 **src/app/app.scss**中：

```
.logo {
    max-height: 39px;
    margin-top: 6px;
}

ion-menu ion-content {
    background-image: url('../assets/images/login-background.png');
    background-size: cover;
    background-position-x: 40%;
}

ion-menu scroll-content {
    margin-top: 44px;
}

ion-menu .item {
    background-color: transparent;
    color: #fff;
}
```

我们稍微修改了logo的位置，给整个登录页加了个背景图，我们给菜单设置了一个边距，这样在真机上运行的时候显示内容看起来不会遮住状态栏，我们也把菜单项变透明了。
接下来，我们将修改登录页和主页，给他们添加一些自定义样式。
> 修改 **src/pages/home/home.scss**如下：

```
page-home {

    ion-label {
        white-space: normal;
    }

    ion-row {
        padding: 0;
        margin: 0;
    }

    ion-col {
        padding: 0;
        margin: 0;
    }

    toolbar-content {
        padding: 0;
        margin: 0;
    }
}
```

这里没啥新奇的，我们只是修改了一些布局元素的排位，这样他们看起来漂亮些。

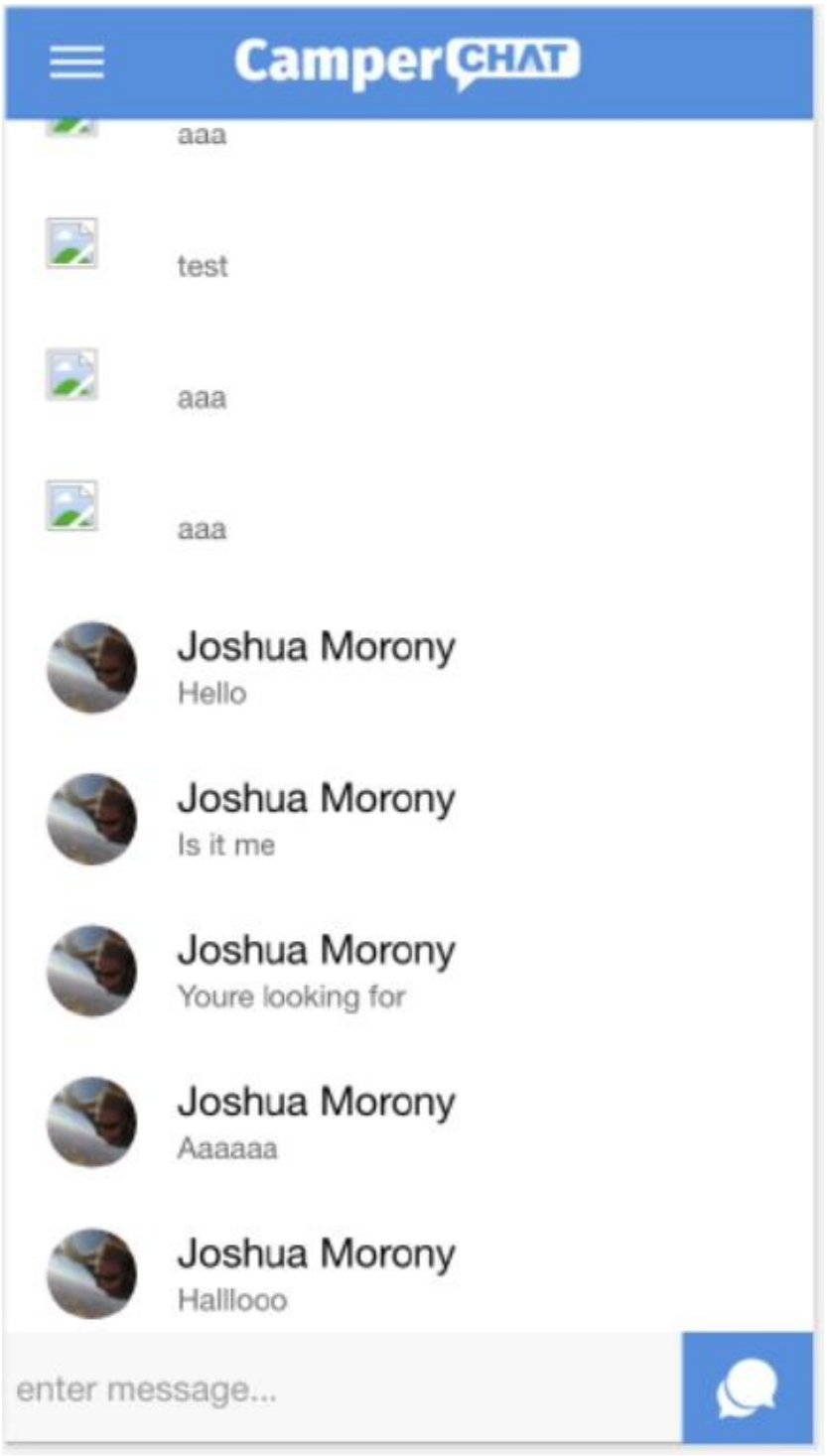
> 修改 **src/pages/login/login.html**为如下：

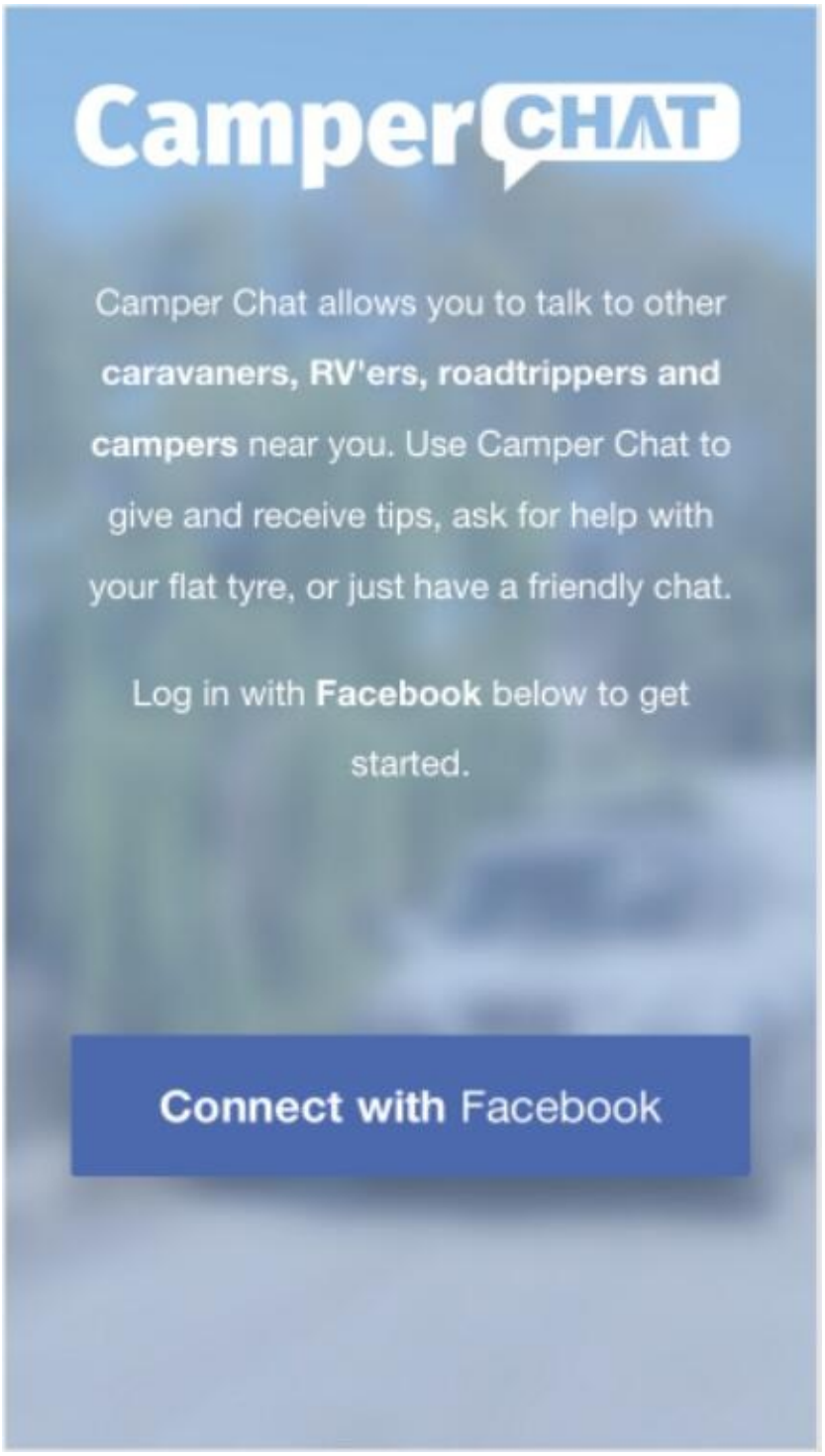
```
<ion-content padding>
<ion-row>
<ion-col>
<img src="assets/images/logo.png" class = "login-logo"/>
</ion-col>
</ion-row>
<ion-row class="login-description">
<ion-col>
<p>Camper Chat allows you to talk to other <strong>caravaners,
RV'ers, roadtrippers and campers</strong> near you. Use
Camper Chat to give and receive tips, ask for help with your
flat tyre, or just have a friendly chat.</p>
<p>Log in with <strong>Facebook</strong> below to get started.</p>
</ion-col>
</ion-row>
<ion-row>
<ion-col>
<a (click)="login()"><img src="assets/images/facebook-button.png" class="login-button"
 /></a>
</ion-col>
</ion-row>
</ion-content>
```

> 修改 **src/pages/login/login.scss** 文件如下：

```
page-login {
    .scroll-content {
        background-image: url('../assets/images/login-background.png');
        background-size: cover;
        background-position-x: 40%;
        text-align: center;
    }

    .login-description {
        height: 60%;
        line-height: 2em;
    }

    .login-description p {
        color: #fff;
    }

    .login-button {
        -webkit-box-shadow: 0px 6px 20px 0px rgba(0,0,0,0.18);
        -moz-box-shadow: 0px 6px 20px 0px rgba(0,0,0,0.18);
        box-shadow: 0px 6px 20px 0px rgba(0,0,0,0.18);
    }
}
```

这次我们给登录页模板添加了一些额外的类。在这个*.scss*文件中我们有设置了一个背景图（和登录页一样的）但是这次是应用到菜单去的。我们给描述内容进行了一些定位同时给Facebook登录按钮添加了一点阴影。
这就是基本样式的全部了，现在你的应用看起来应该是这样的：


CamperCHAT
aaa
test
aaa
aaa
Joshua Morony
Hello
Joshua Morony
Is it me
Joshua Morony
Youre looking for
Joshua Morony
Aaaaaa
Joshua Morony
Hallllooo
enter message...



我们还有一个最有趣的部分需要去做，就是添加一些动画。

创建动画

需要记住一个重要的事情使用**translate3d**属性去给元素制作动画（即使是用来制作2D动画）。通过使用这个属性，例如我们去操作left属性，就会调用设备的GPN(Graphics Processor Unit图形处理单元)。他会调用硬件加速来进行动画并且更流畅。你可能会发现在桌面浏览器上运行很流畅但是在真机上运行的时候效果却不是很好，所以使用硬件加速非常重

要（尽管依赖GPU太多的话会有其他缺点，例如，电池掉太快）。
制作动画的话我们就要定义我们自己的**keyframes**。基本上，他让你指定动画过程的特定场景是的具体属性。例如：

```
@-webkit-keyframes slideInSmooth {
    0% {
        -webkit-transform: translate3d(-100%,0,0);
    }
    100% {
        -webkit-transform: translate3d(0,0,0);
    }
}
```

我们这里将的是动画开始(0%)元素应该向左偏移100%（这三个参数在translate3d中代表x，y和z轴），所以他应该是在屏幕外。然后在动画结束(100%)的时候元素应该回到了正常的位置。
我们可以这定义一个类似的哦那个话：

```
@-webkit-keyframes slideInSmooth {
    0% {
        -webkit-transform: translate3d(-100%,0,0);
    }

    50% {
        -webkit-transform: translate3d(50%,0,0);
    }

    100% {
        -webkit-transform: translate3d(0,0,0);
    }
}
```

这样元素会从左边开始，然后回到右边，最终回到正常位置。你可以随你喜欢定义大量这样的中间步，唯一的重点是你只能有一个0%和100%（也只能是其中的值）否则，动画就无效。
我们只要创建一个类使用这个动画，然后给一个元素添加这个类就可以将动画附加到他上面去了。我们将不会去使用这些动画，我们会为用户照片和他们的信息创建一些类似的动画。
> 修改 **src/app/app.scss**为如下：

```
@-webkit-keyframes animateInPrimary {
    0% {
        -webkit-transform: translate3d(-100%,0,0);
    }

    100% {
        -webkit-transform: translate3d(0,0,0);
    }
}

@-webkit-keyframes animateInSecondary{
    0% {
        opacity: 0;
    }

    50% {
        opacity: 0;
    }

    100% {
        opacity: 1;
    }
}

.animate-in-primary {
    -webkit-animation: animateInPrimary;
    animation: animateInPrimary;
    -webkit-animation-duration: 750ms;
    animation-duraton: 750ms;
}

.animate-in-secondary {
    -webkit-animation: animateInSecondary ease-in 1;
    animation: animateInSecondary ease-in 1;
    -webkit-animation-duration: 750ms;
    animation-duraton: 750ms;
}
```

这里我们创建了两个不同的动画。**animateInPrimary**动画将使元素从左边屏幕外开始，滑动到他的普通位置。**animateSecondary**动画从完全透明到后半程逐步显现出来。
我们将这两个动画分成两个类，动画设置都是750ms。要用上他们的话，我们只需要把**animate-in-primary**和**aniamte-in-secondary**附件到元素上去即可。
> iugai src/pages/home/home.html的列表如下：

```
<ion-list no-lines>
    <ion-item *ngFor="let message of messages">
        <ion-avatar item-left class="animate-in-primary">
            <img [src]="message.picture">
        </ion-avatar>

        <div class="animate-in-secondary">
            <h2>{{message.username}}</h2>
            <p>{{message.message}}</p>
        </div>
    </ion-item>
</ion-list>
```

看到没，我们给avatar添加了primary animation，给信息区域添加了secondary animation。如果重新加载应用的话就可以看到元素动画了。

结论

恭喜通过了Camper Chat指南的学习。在开发过程中，我们学到了很多，主要是以下几点：

- 导航
- 使用侧滑菜单
- 使用PouchDB存储本地数据
- 使用Cloudant存放远程数据
- 使用Facebook API的验证和其他功能
- 实时更新和展示数据

改进的空间永远都存在，特别是当你学习事物的时候。遵循指导手册固然很好，但是自己去学习弄清楚一些事情就更完美了。希望你有足够的背景知识来自己完成一些功能扩展，以下是一些想法：

- 在收到新信息的时候通过Cordova插件进行震动【简单】
- 添加‘Invite Friends’选项，使用Facebook API的App Invites【中等】
- 添加“Channel Categories”让用户在不同的聊天室之间切换，例如：汽车问题，普通聊天，露营【困难】
- 添加私聊功能【噩梦】

记住，在你学习的过程中，Ionic 2文档是你最好的朋友。

接下来？

现在你有一个完整的应用了，但是这不是故事的结局。你需要让他运行到真实设备上去，提交到应用商店，这不是个简单的任务。本书的最后部分会带你完成这些内容。你可以先看看。

第八章：测试&调试

测试 & 调试

当你在制作应用的时候，有点小错误和发生点小报错的啥的是很正常的。有时候错误不明显，所以知道怎样去追踪错误是非常重要的。
本节课将会研究如何最好的去调试Ionic 2应用，但是不会讲什么是调试或者如何使用调试工具（例如，查看源代码，设置断点，监听网络请求等等）。如果你不熟悉基于浏览器的Javascript调试，那么强烈建议一定要先你看看这个。

浏览器调试

开发应用的时候第一发通常是桌面浏览器。能够通过浏览器直接调试甚至实时加载是个很厉害的事情，你可以马上就看到你的代码变更迭代非常快。
重要的一点是需要时刻记住他不代表在真机上的运行效果。大部分情况下，在浏览器上什么效果，在真机上就是什么效果，但是有可能一些Cordova插件的原因会有所不同，因为他们不能直接在浏览器里面测试。
正确的方法是在浏览器里面测试个大概，当所有东西测试得差不多的时候，在到真机上面测试以使所有事物运行正常。

iOS调试

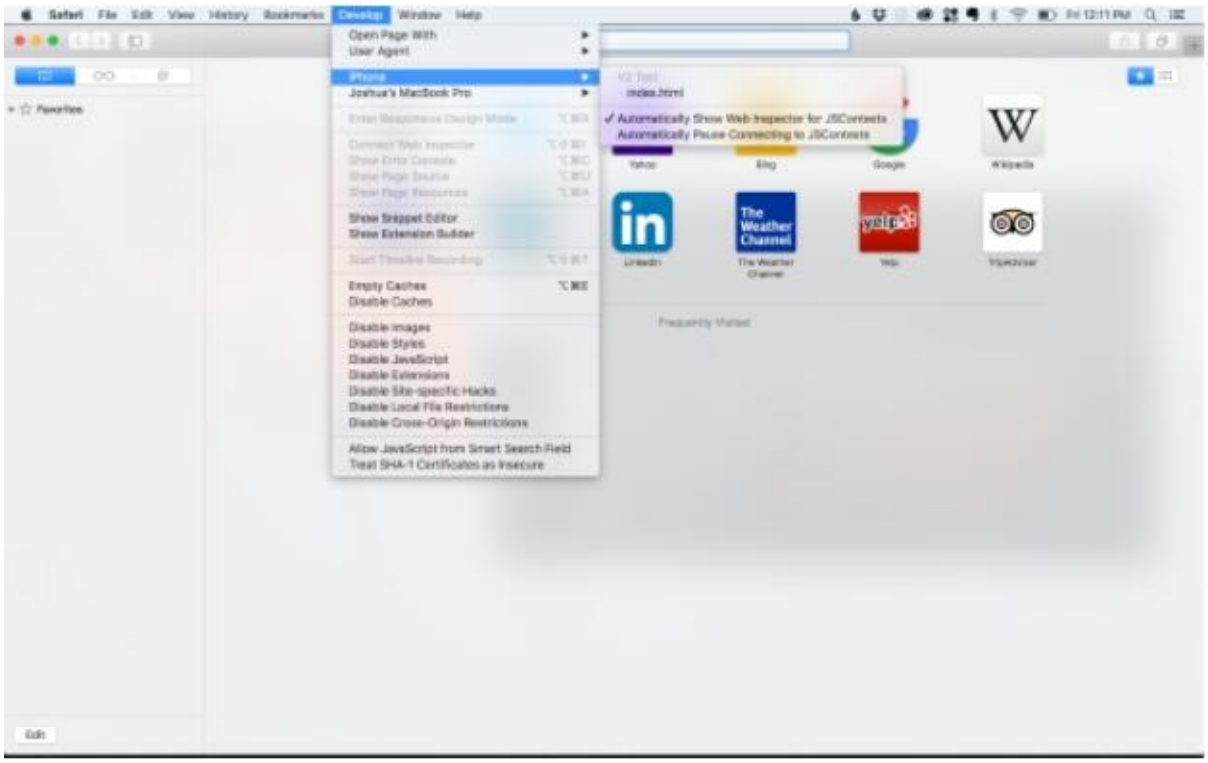
想要在真实iOS设备上调试的话，先得运行如下命令：

```
npm -g install ios-sim ios-deploy
```

他将会安装iOS模拟器（用来做电脑上模拟iOS设备）和*ios-deploy*将用于在把应用通过USB部署到设备上去。
一旦设置完成，只需要在项目目录下运行这个命令：

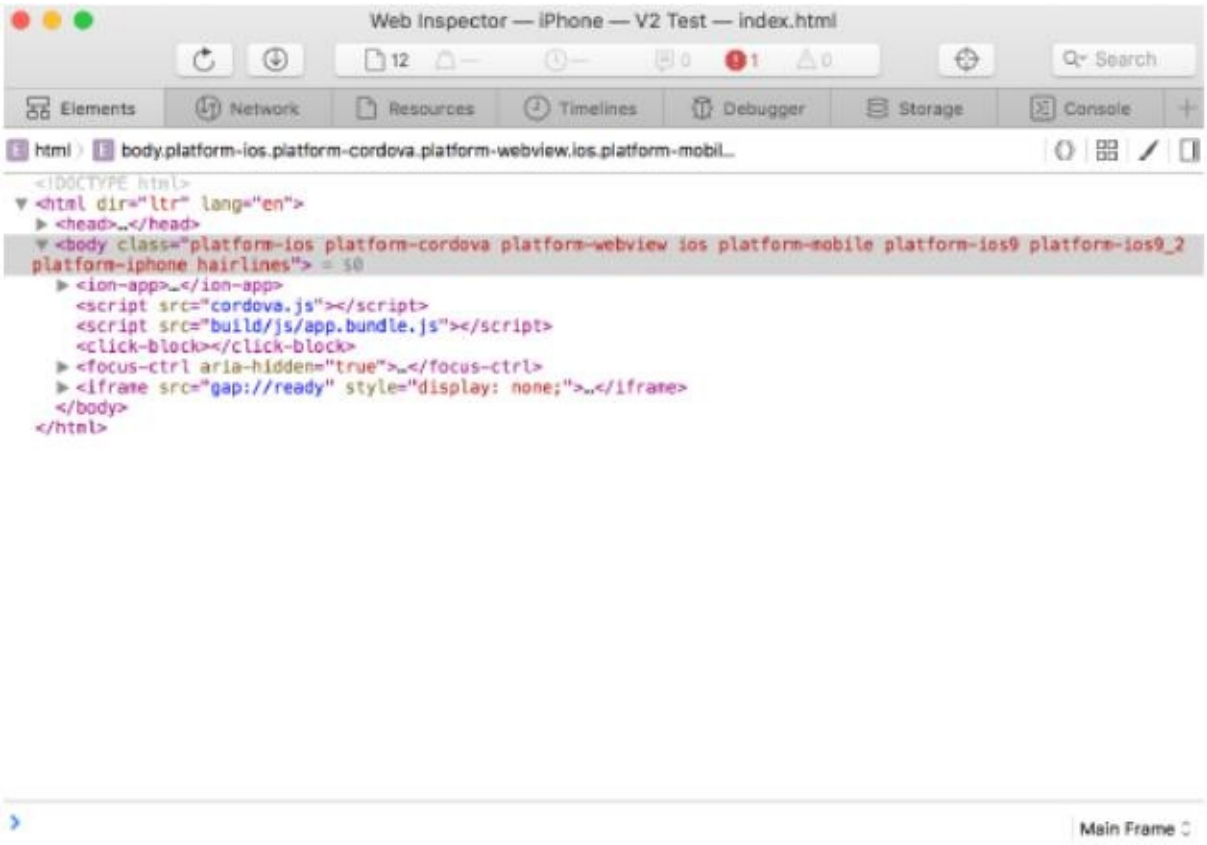
```
ionic run ios
```

应用就会被部署到你的设备上去（没有设备的话就是部署到模拟器上）。如果你是第一次运行此命令，那么可能要运行两次才能正常工作。
一旦应用在设备上运行起来，你就可以在电脑上通过Safari Dev Tools来调试了。只要打开**Safari**然后去到：
Develop > iPhone > index.html



如果Develop菜单没有出来的话，你需要先去菜单栏的menu启用他：*Safari* > *Preferences* > *Advances* > *Show Develop*

一旦在Safari中打开了**index.html**可以看到这样的调试画面：



重要：这个方法只会在Mac上有用。如果你没有Mac的话，那么你可以通过GapDebug来安装

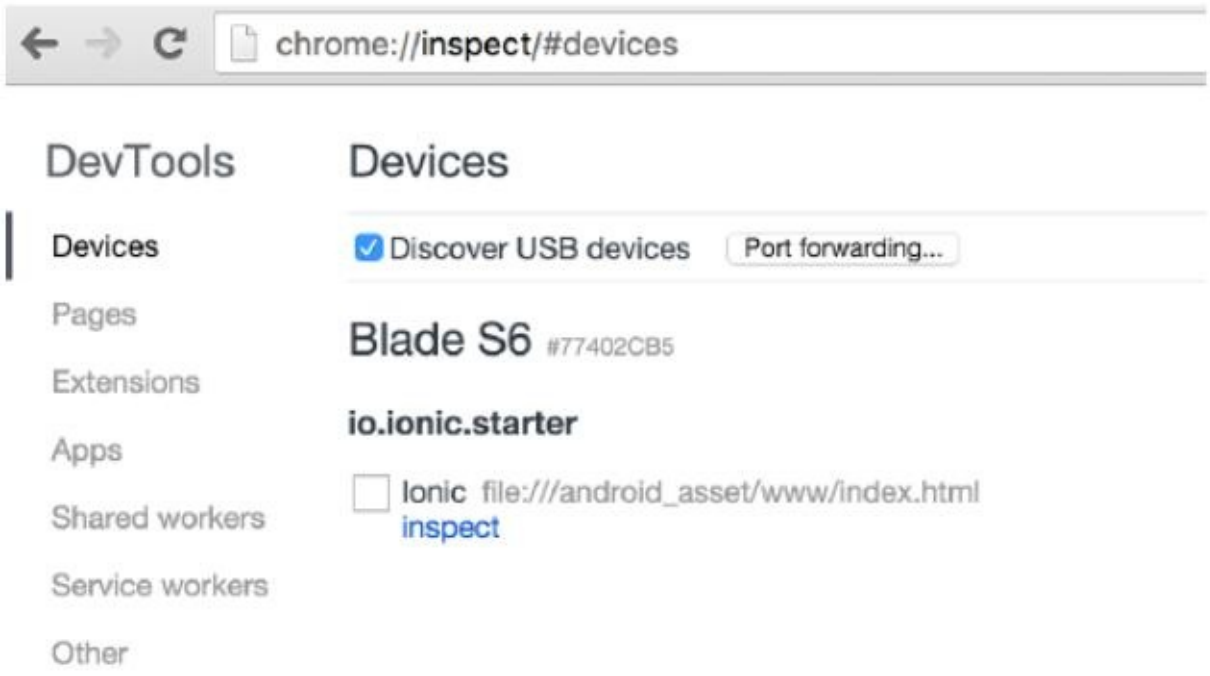
和调试iOS应用。用这个方法的话首先需要生成应用的*.ipa*（我们会在构建**iOS**应用课程里面讨论）然后直接安装他。
使用**Test Flight**测试最终构建办也是个不错的方法。

Android 调试

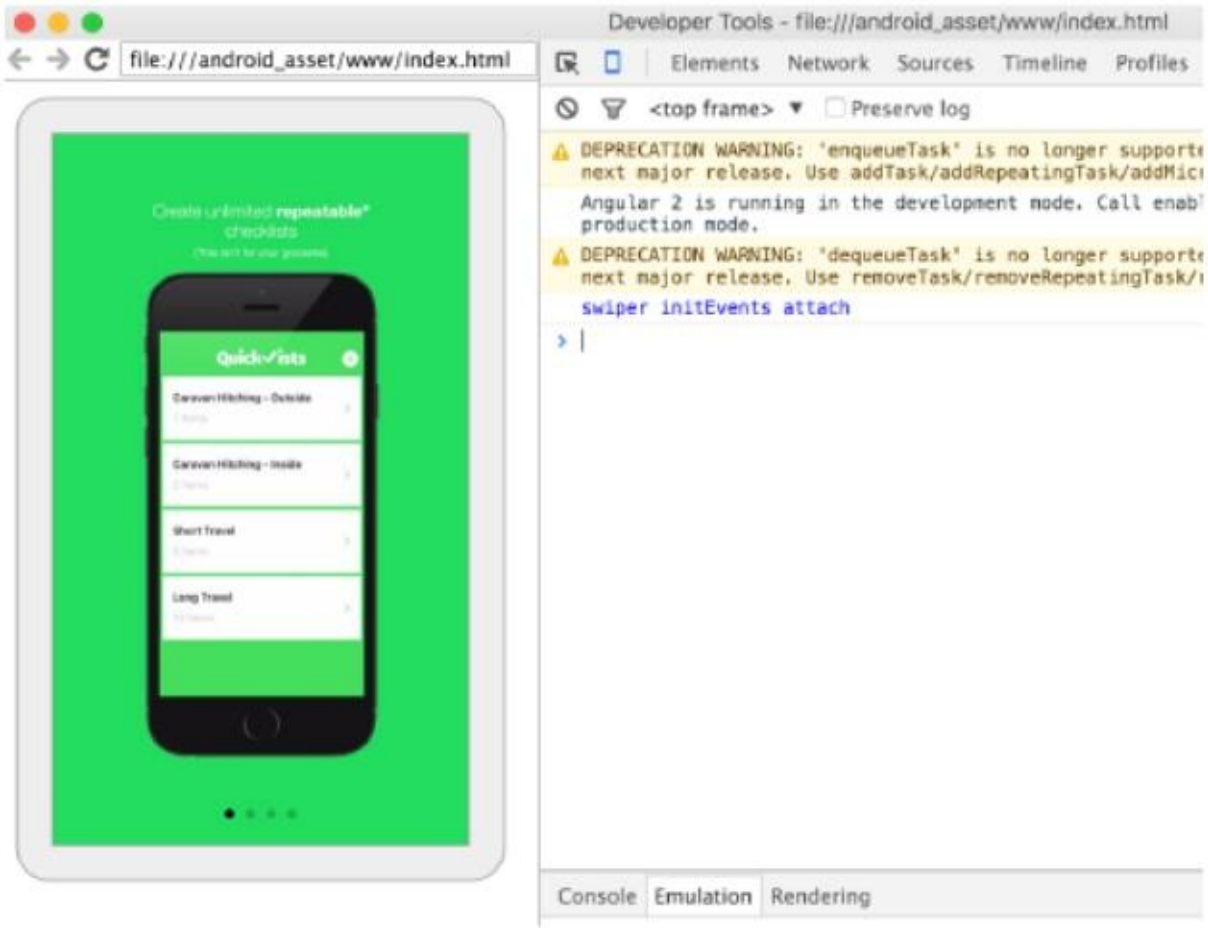
在Android上调试跟添加需要调试的目标设备一样简单，运行：

```
ionic run android
```

然后在你的桌面浏览器（Chrome）中可以去这个地址：*chrome://inspect/#devices*



然后点击需要调试的设备的'Inspect'，我们就可以看到调试工具了：



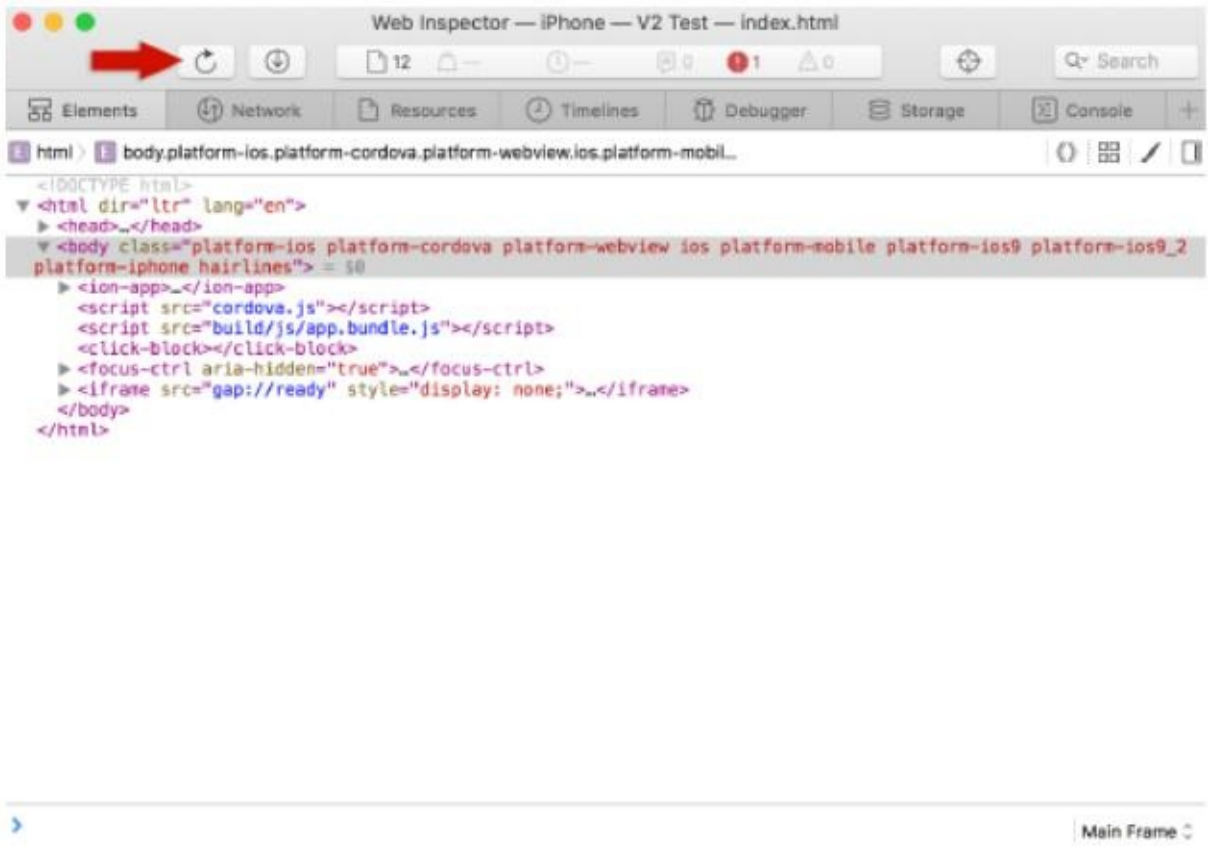
重点：需要激活设备上的通过USB调试选项。每个设备上激活USB调试的方法可能不大一样，所以尽管去Google“你的设备名字 激活usb调试”就可以了。

贴士 & 普通错误

调试应用的难易度基本来自你多年的开发经验累积，这里给你一些调试的小技巧和小贴士来帮助你调试Ionic 2应用。

启动白屏或者一片空白？

这种事情经常发生，原因通常是在设备上启动的时候发生错误。有时候你启动应用之看到一片空白，即使是打开调试工具也看不到错误发生。这可能是因为在启动调试器之前就报错了，然后break【译者：跳出？不大合适】了应用。所以，如果你看到类似的情况，确保调试器准备好了，然后点击调试器的刷新按钮，在Safari中，是这个样子的：



他会在准备好了调试器的情况下重新加载应用，意味着你不会漏掉任何一个开始的错误。90%的情况下会发现一些Javascript错误，修改之后就应用就可以正常工作了。

调试器

希望你知道断点对调试应用多么重要，如果不知道的话，我再次强烈建议你读一下这个。你可以打开'Source'标签页手动添加一些断点，但是你也可以在任何地方加这个小小的关键字：

```
debugger;
```

他会使应用在这个点暂停。（同时会打开Dev Tools）

404错误

你的ap是不是在浏览器中正常工作但是在真机上运行的时候遇到404错误？确认一下是否包含一Whitelist插件：

```
ionic plugin add cordova-plugin-whitelist
```

以及在**index.html**中是否有一个正确的Content Security Policy（CSP）元标签：

```
<meta http-equiv="Content-Security-Policy" content="font-src * data:; img-src
    * data:; default-src * 'unsafe-eval' 'unsafe-inline'">
```

真机上插件不工作

第一件事情肯定是确保安装了插件：

```
ionic plugin add [plugin name]
```

说起来可能有点蠢，但是需要双重检查保证安装成功：

```
ionic plugin ls
```

这个命令将会列出当前已安装的所有插件，我还是经常会忘记安装插件。如果你安装好了插件，也要确保没有过早去访问插件。如果你在设备住呢比好之前去尝试访问插件的话，也就是在这个之前：

```
platform.ready().then(() => {
});
```

那么肯定是不正常工作的。最后，如果你用尽所有办法依然无法工作的情况下，你可以试着移除插件：

```
ionic plugin rm [plugin name]
```

然后重新添加插件：

```
ionic plugin add [plugin name]
```

调试程序是一个苦难且令人沮丧的任务，特别是在真机上调试的时候，但是随着开发经验的增加，你就更容易感觉到问题出现在哪里，开发流程会越来越简单。如果你卡在一个错误上的话，可以随时去Ionic 论坛求助，只要尽量详细提供错误信息以及你尝试过的方法。

通过GapDebug安装应用

如果你是使用PhoneGap构建的，并且有*.ipa*或者*.apk*文件那么你就可以使用GapDebug对他们进行测试。有很多方法可以安装他们 -- 可以用iTunes来安装 .ipa 文件或者你可以使用**adb**来安装 .apk 文件 -- 但是用GapDebug就简单多了，GapDebug也提供了大量牛逼的调试工具（尽管如此，如果你在找更高级的调试方法我建议你使用**adb**）。去这个网站下载和安装

GapDebug就可以了。安装完成后，系统托盘上就可以看到了。
在你安装好GapDebug之后，先确保通读**GapDebug First-Time Configuration and Setup** --因为在让GapDebug和你的iOS或者Android设备正常协作之前有一些事情需要去做。
所以东西设置好之后就可以把设备连接到电脑了，打开GapDebug，将应用文件（.ipa或者.apk）通过GapDebug拖放到设备中。这不仅是在设备上安装应用超快的途径同时他给你提供了一些非常有用的调试工具（实际是跟你已经熟悉了的浏览器调试工具一样的）。一旦通过GapDebug安装好了你的应用，直接在GapDebug里面点击应用就可以调出调试工具了。

第九章：构建与提交

准备素材

此刻你的应用应该打包好了，正常工作，可以提交到应用商店里。在将他提交到应用商店之前，我们还有一些需要做的事情。
你可以在阅读本书的同时在制作自己的应用，前提是你已经完成了其中一个范例应用的制作。这样的话，我还是建议你通过这些步骤，或者至少读完他们，这样你就学习提交过程是怎样的。然后我希望你在将应用实际提交到应用商店之前暂停一下，我很高兴你在你的应用中很好的用上了这些胆码，但是你不能提交这些范例中的任何一个到应用商店。

生成图标和闪屏

市面上有大量的设备，他们有着不同的屏幕尺寸和分辨率，在制作图标和闪屏的时候，我们需要去适配这些设备。
头脑正常的人都不会在给同一个文件制作50种不同版本而觉得亦可赛艇。幸运的是Ionic提供了一个非常简单，接近全自动的方法。我们可以通过*ionic resources*命令来为我们生成这些。虽然Ionic给你做了很多事情，但是他不会为你设计你的图形资源，所以我们需要做一些设置。

设计图标

Ionic资源工具之要求一个 192x192 的图标作为基本，但是由于应用商店需求一个大一些的图标，因此最好先设计为 1024x1024（甚至是 2048x2048）。在大部分时候，将大图缩小比较好，特殊情况下你可以特殊设计小号图标。
> 创建一个 **192x192** 的图标名为 **icon.png** 将他保存到 **resource** 文件夹内（覆盖已有的图标）

设计闪屏

图标很好，因为他们都是方形的。闪屏就相当不幸了，任何形状，任何尺寸。这意味着要么你进行弹性设计，要么每个不同屏幕单独制作一个闪屏。
什么是“弹性”设计？好吧，我管能够和Ionic的资源工具完美工作的设计叫做弹性设计。如果你将闪屏设计为 2208x2208 那么所有的闪屏都可以很好的生成。
问题是永远都会发生不同程度的裁切 -- 有的屏幕比较高有点屏幕比较宽。为了避免设计的重要部分被切掉，尽量将重要部分设计为靠近图片的中心部分，靠近边缘的是稍微不那么重要的东西（即，背景图片部分）。
> 从下载包中获取**splash-screen-template.psd**或者**splash-screen-template.png**文件。基于他们制作自己的 **2208x2208** 闪屏然后存放到**resources**文件夹命名为 **splash.png**。

运行资源工具

现在制作好了基本图标和闪屏，可以通过如下命令生成其他的图标和闪屏：

```
ionic resources
```

设置Bundler ID和App Name

在进行构建之前有一个重要的步骤是修改**config.xml**文件里的App Name和Bundle ID。这个文件的前几行是这样的：

```
<?xml version='1.0' encoding='utf-8'?>
<widget id="io.ionic.starter" version="0.0.1"
    xmlns="http://www.w3.org/ns/widgets"
    xmlns:cdv="http://cordova.apache.org/ns/1.0">
<name>V2 Test</name>
<description>An Ionic Framework and Cordova project.</description>
<author email="hi@ionicframework" href="http://ionicframework.com/">Ionic Framework Te
am</author>
```

你应该把上面这些配置改成你自己的，包括**id**也就是现在的**io.ionic.starter**。这个值应该匹配你下节课签名的值，应该被设置为类似：**com.yourname.project**。如果你是第一次发布此应用的话，版本好最好也设置为**1.0.0**。

设置Cordova偏好

这本地上不是一个什么步骤，但是你可以在**config.xml**中进行一些偏好设置来影响如何构建。**config.xml**文件有一些默认的如下：

```
<preference name="webviewbounce" value="false" />
<preference name="UIWebViewBounce" value="false" />
<preference name="DisallowOverscroll" value="true" />
<preference name="android-minSdkVersion" value="16" />
<preference name="BackupWebStorage" value="none" />
```

但是还有成吨的偏好设置选择，具体查看此处

缩减素材

这也是一个可选步骤但是我个人觉得很重要。如果你想尽量减少最终发布版的大小，其中一个方法就是通过一些工具来缩减图片尺寸例如TinyPNG。只要通过工具跑一遍图片然后重新保存到项目中即可。

在Mac或者PC上为iOS应用签名

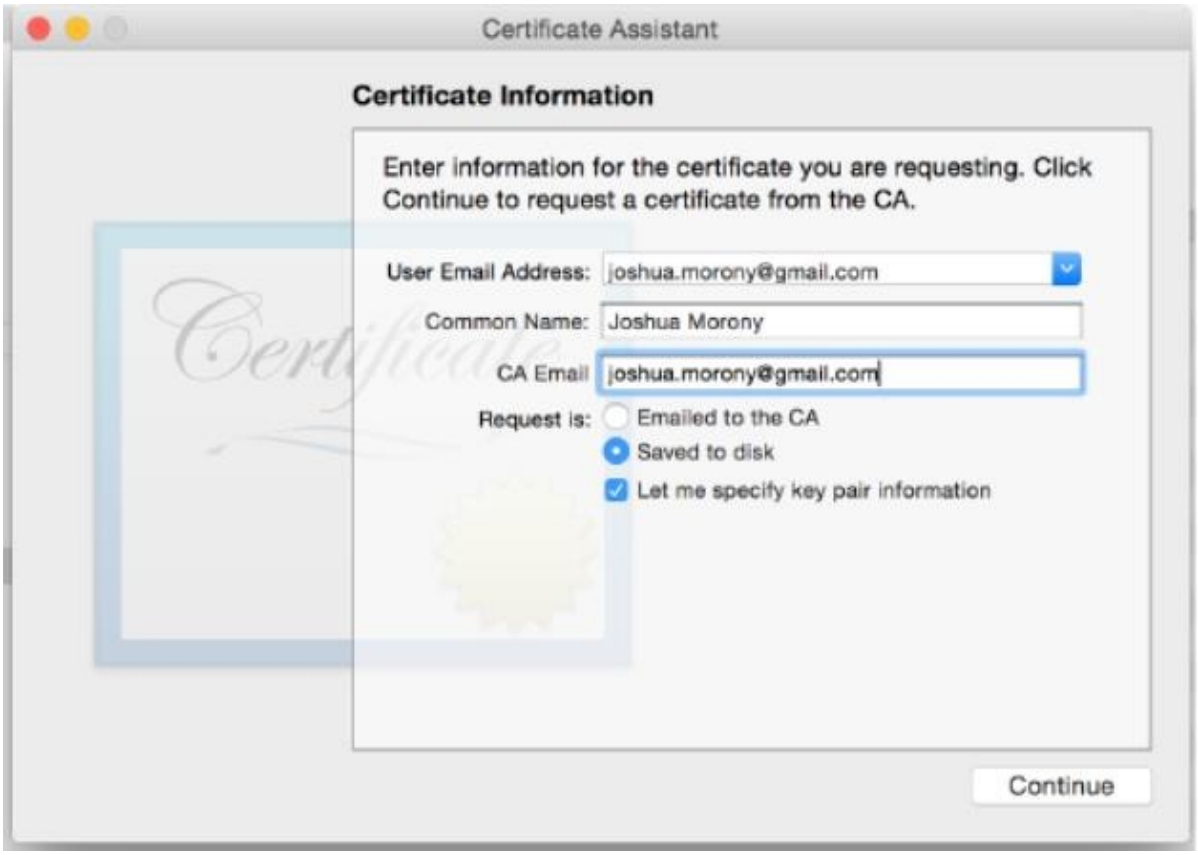
iOS和Android应用都需要'签名'。理念和普通的签名是一样的，用来证明你授权了并有权限来做这些。平台之间的签名工作有些不同。
iOS需要一个**.p12**文件，这个文件是一个key组成和一个开发或者分发证书组成的。对于**.p12**文件需要创建一个配置文件。**.p12**文件的目的是识别你是一个iOS开发人员，配置文件用于识别应用，服务和允许安装此应用的设备。准备好这些东西还是需要点工作流的，如果一步一步遵循这个引导来做的话不会有什么问题。
如果你有Mac的话，你可以使用XCode来自动完成其中一些步骤，但是我会一步一步的讲解这样在你没有Mac的情况下你可以备用方法了。
开始之前你得报名iOS Developer Program

在Mac上签名iOS应用

如果你有Mac的话，按照这些步骤，如果没有，跳过此部分去阅读在**Windows**上签名**iOS**应用。在开始之前先确定你安装好了XCode。
生成一个证书签名请求

- 打开**Keychain Access**
- 去**Keychain Access > Preferences:**

- 去**Certificates**标签页然后确保所有选项都关闭了
- 现在去**Keychain Access > Certificates Assistant > Request a Certificate From a Certificate Authority**
- 填写弹出框，选择**Saved to disk**和**Let me specify key pair information**：

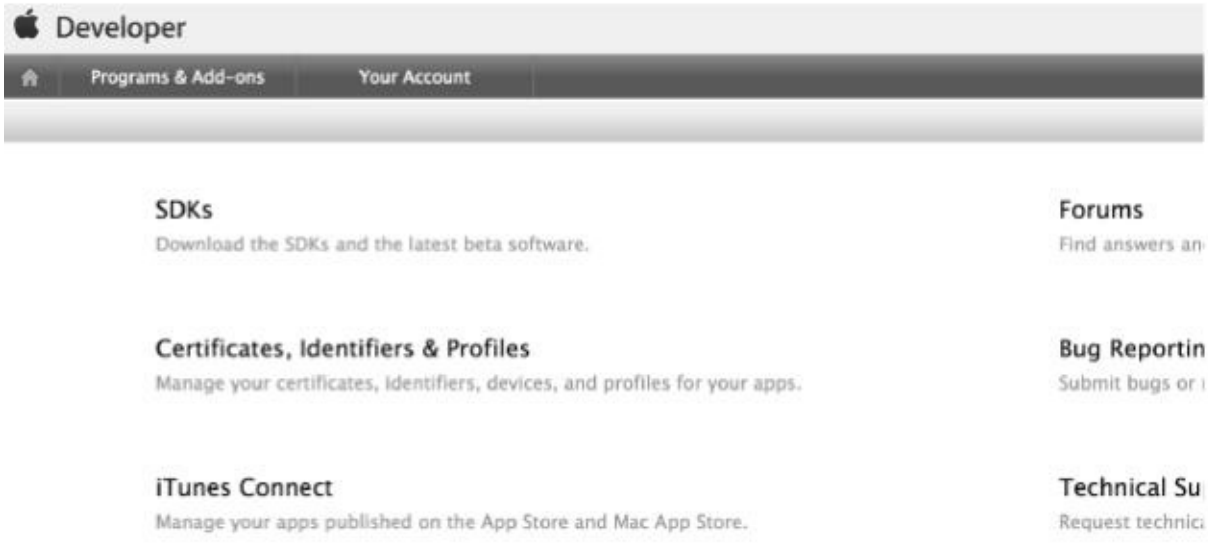


点击继续然后选择一个位置将文件存放到本机。下一个屏幕中选择**2048 bits**和**RSA**：

证书签名请求现在就会保存到你指定的文件夹中，你应该注意到Keychain Access中会生成一个公钥和私钥对。

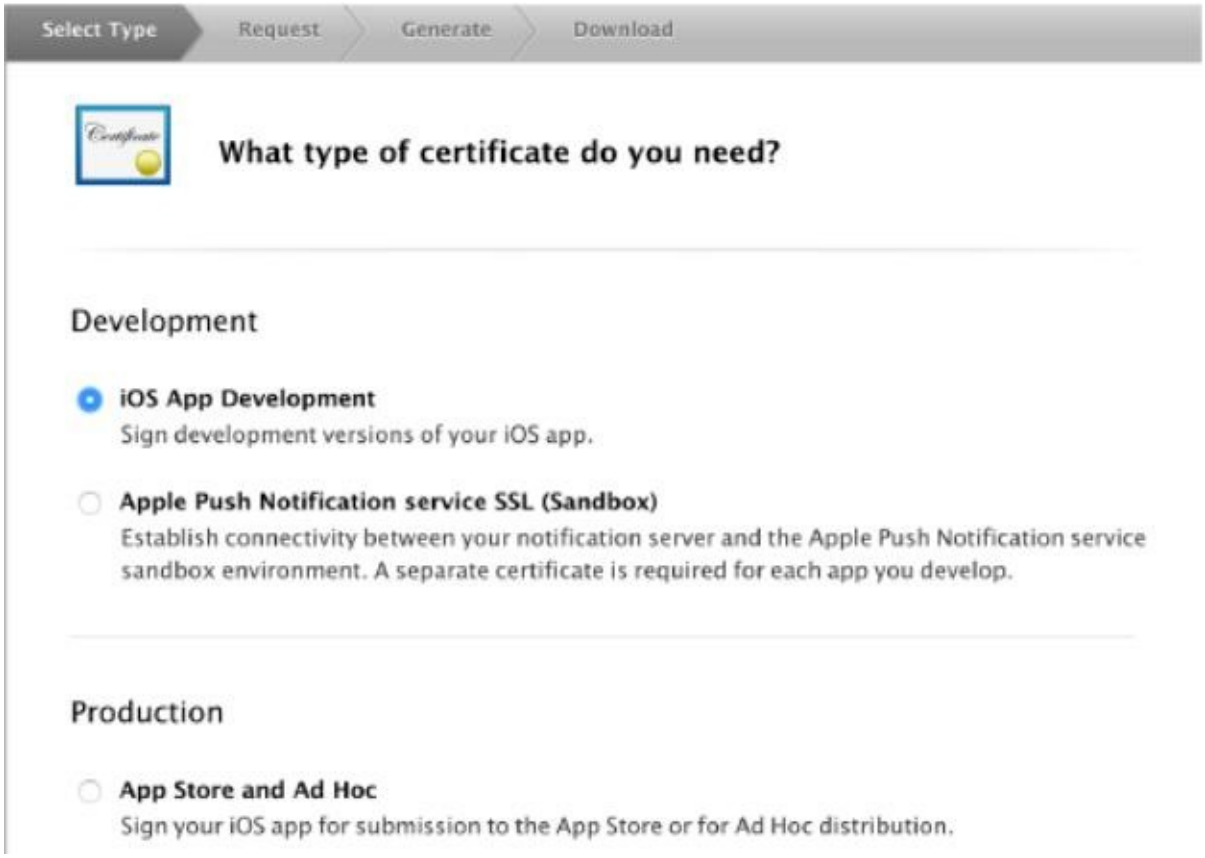
制作证书

要完成后续的步骤的话需要登录到Apple Member Center然后去**Certificates,Identifiers & Profiles**部分：



- 选择**iOS Apps**下面的**Certificates**，点击右上的 **+** 按钮：

接下来，选择需要生成的证书类型。如果你想要给测试应用生成证书那么选择**iOS App Development**，如果是应用商店的发行版的话，那么选择**App Store and Ad Hoc**。



可能你注意到了屏幕上的其他选项。如果你想要使用类似Apple Push Notifications服务或者WatchKit的话，你需要另外创建证书，但是我们用不上。

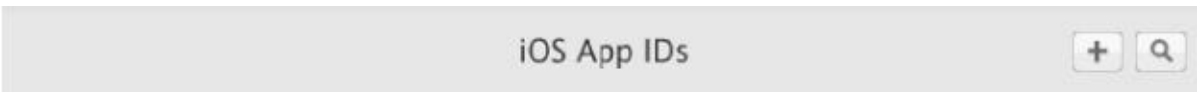
- 选择证书类型然后点击**Continue**
- 现在就要你刚才通过Keychain Access创建的证书签名申请了，点击继续然后上传签名申请。一旦选择了**.certSigningRequest**文件点击**Generate**。

现在你可以下载你的证书了。将他下载到一个安全的地方，然后打开安装。

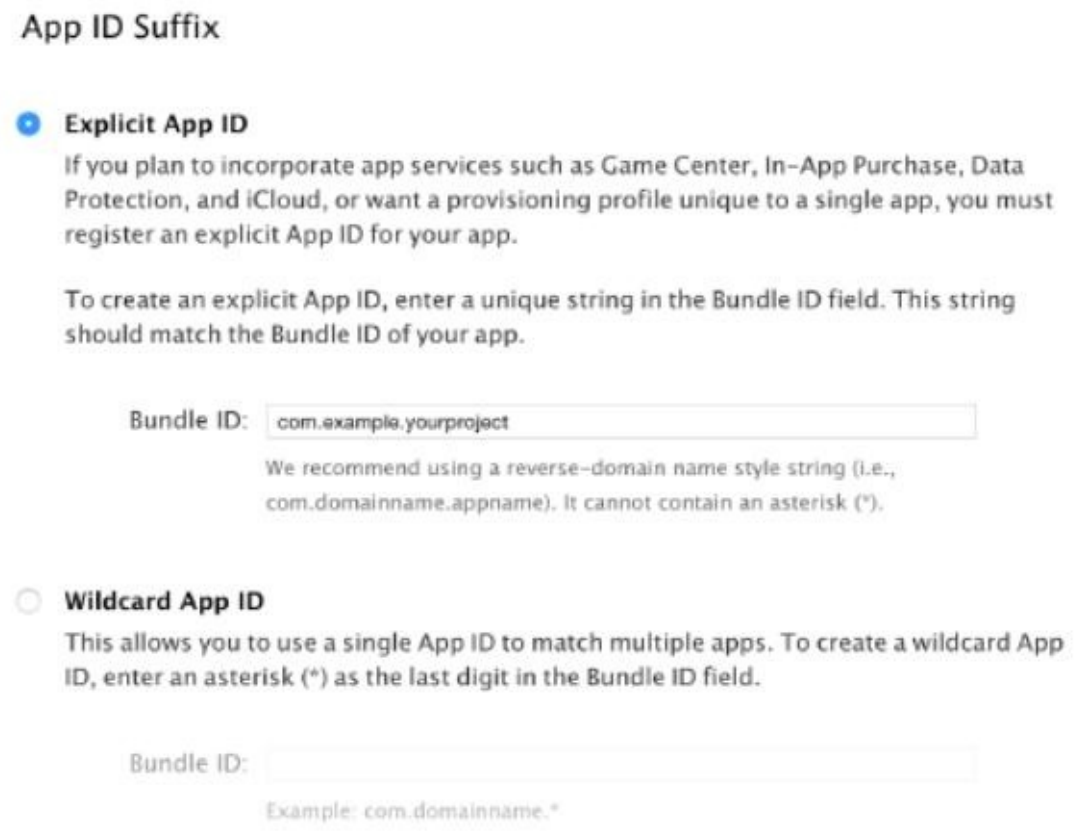
制作**Identidfier**

如果你是使用XCode管理应用的话，那么就不需要关心这些步骤来，因为XCode可以使用一个Wildcard App ID，你也可以手动创建你自己的App ID（这里创建的Bundle ID必须和config.xml文件里面的一样）。以下是步骤：

- 点击**App IDs**然后的点击 **+** 图标



- 填写App ID Description然后提供一个Explicit App ID就像这样：



这个必须和**config.xml**里面的**id**一致。

- 来到屏幕底部点击**Continue**
- 下一个屏幕点击**Submit**

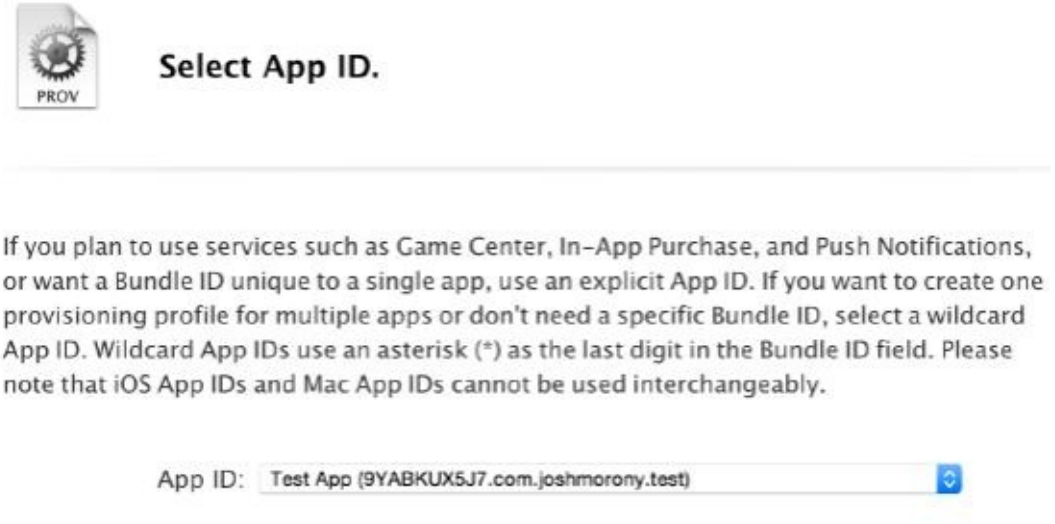
此时App ID注册完成，可以在配置文件中使用了。

创建一个配置文件（**Provisioning Profile**）

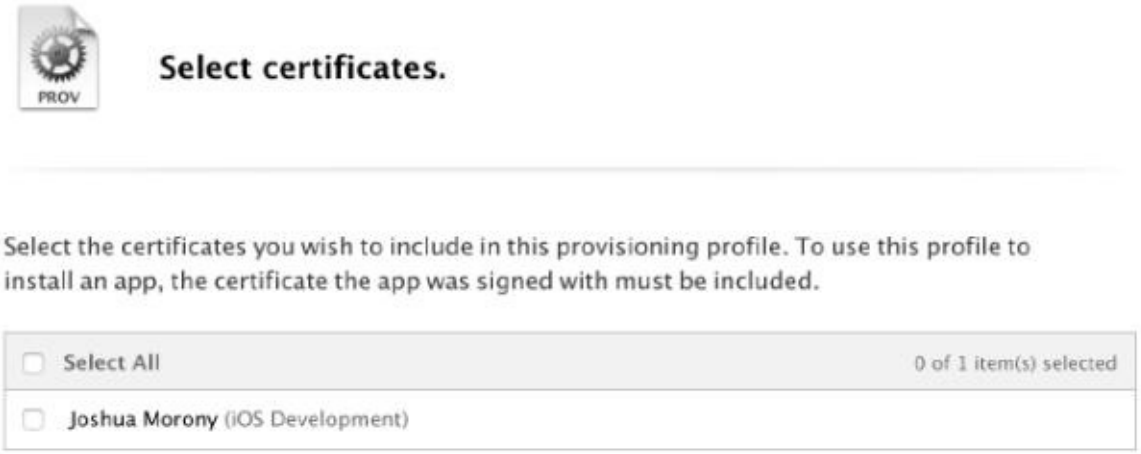
创建配置文件之前需要添加应用可以运行的设备。

- 点击**Devices**然后点有右上的 **+** 按钮

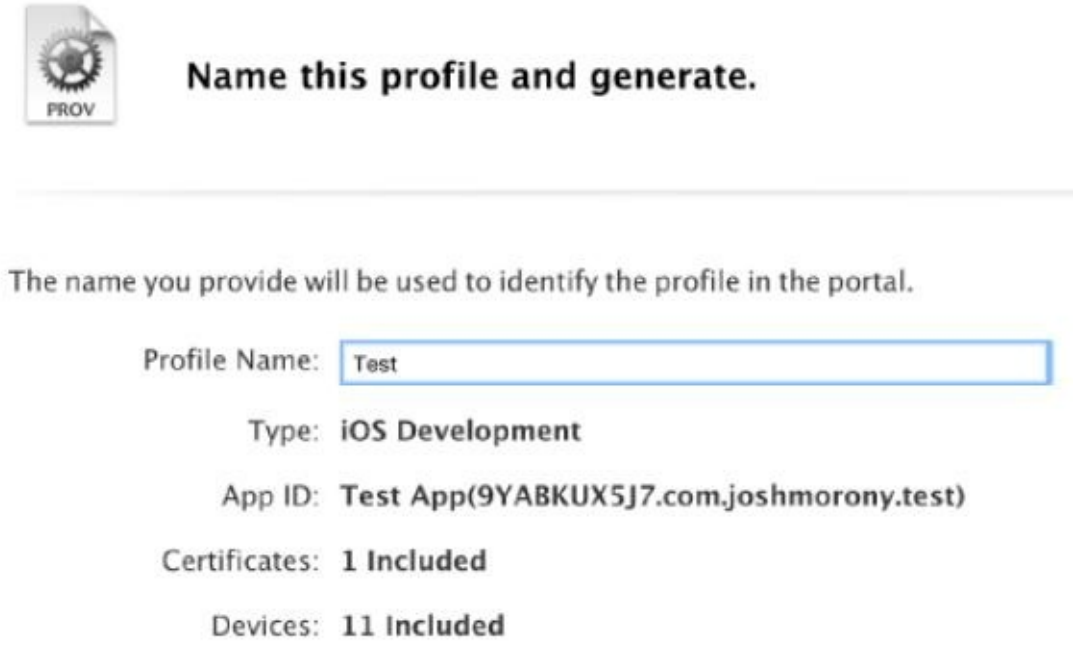
- 如果是给测试制作配置文件的话选择**iOS App Development**，如果是发布版的话那么选择**App Store**。点击**Continue**
- 选择刚才创建的App ID然后点击**Continue**



- 选择想要用的证书然后点击**Continue**



- 选择所有可以想要运行的设备然后点击**Continue**：
- 给配置文件提供一个名字然后点击**Generate**



现在你应该可以下载你的配置文件了。

生成 .p12

*.p12*文件（和配置文件一起的）是类似PhoneGap Build这样的服务必需的。如果你有一台Mac的话，这个跟你没什么关系，尽管如此你还是可以使用类似PhoneGap Build这样的服务（我不建议）。如果你需要在你的Mac上创建一个 .p12 文件的话。以下是步骤：

- 打开Kaychain Access
- 回到创建证书签名申请的第一步，我提到过 Keychain Access 会生成一个公钥和私钥对。在**Keys**部分找到这个：

可以看到私钥也有了你在iOS Member Center创建的证书了。

- 同时选中私钥和证书，右键点击并选择**Export 2 items...**：

- 然后你会被问到文件存放到哪里，并且你可以选择给**.p12**输入一个密码或者直接留白。也有可能需要输入电脑的管理密码。

恭喜！你现在有**.p12**文件了。

在Windows上签名iOS应用

要完成以下步骤的话要先下载和安装OpenSSL。OpenSSL网址在此：
https://www.openssl.org/source/
下载操作系统适合的版本。

生成一个证书签名请求

OpenSSL安装设置好了之后我们就可以用它来生成一个Certificate Signing Request。这个将会在iOS Member Center撞见开发和发布证书的时候使用。

- 将当前目录改为OpenSSL的bin目录，例如：

```
cd c:/OpenSSL-Win64/bin
```

- 生成一个私钥：

```
openssl genrsa -out mykey.key 2048
```

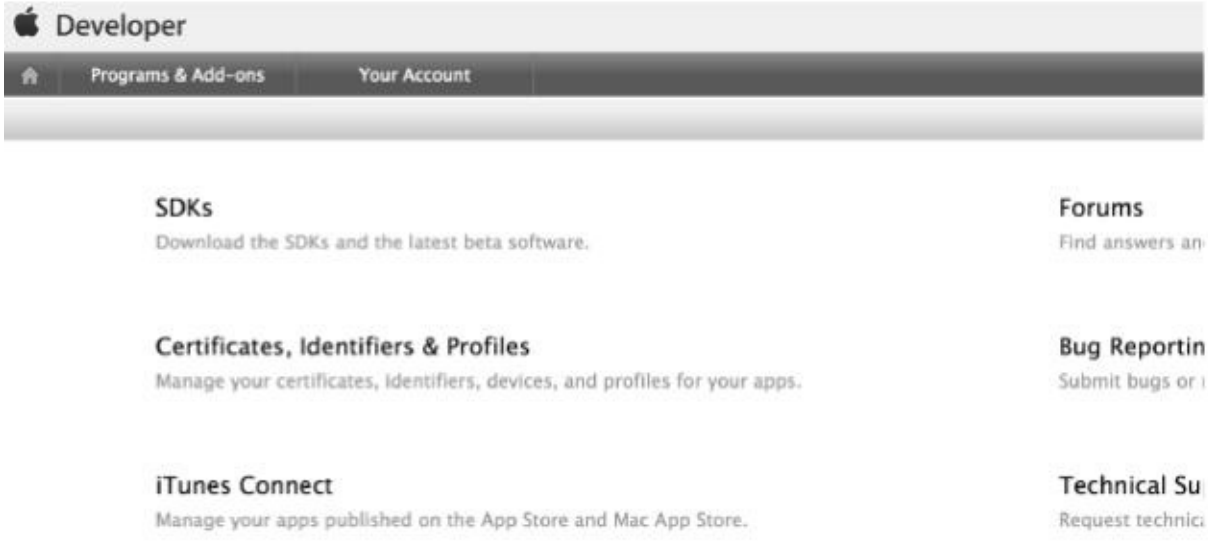
- 使用这个密钥生成一个证书签名申请

```
openssl req -new -key mykey.key -out myCSR.certSigningRequest -subj "/emailAddress
=you@yourdomain.com, CN=Your Name, C=AU"
```

确保用你自己的信息替换邮件地址，名字以及国家代码。

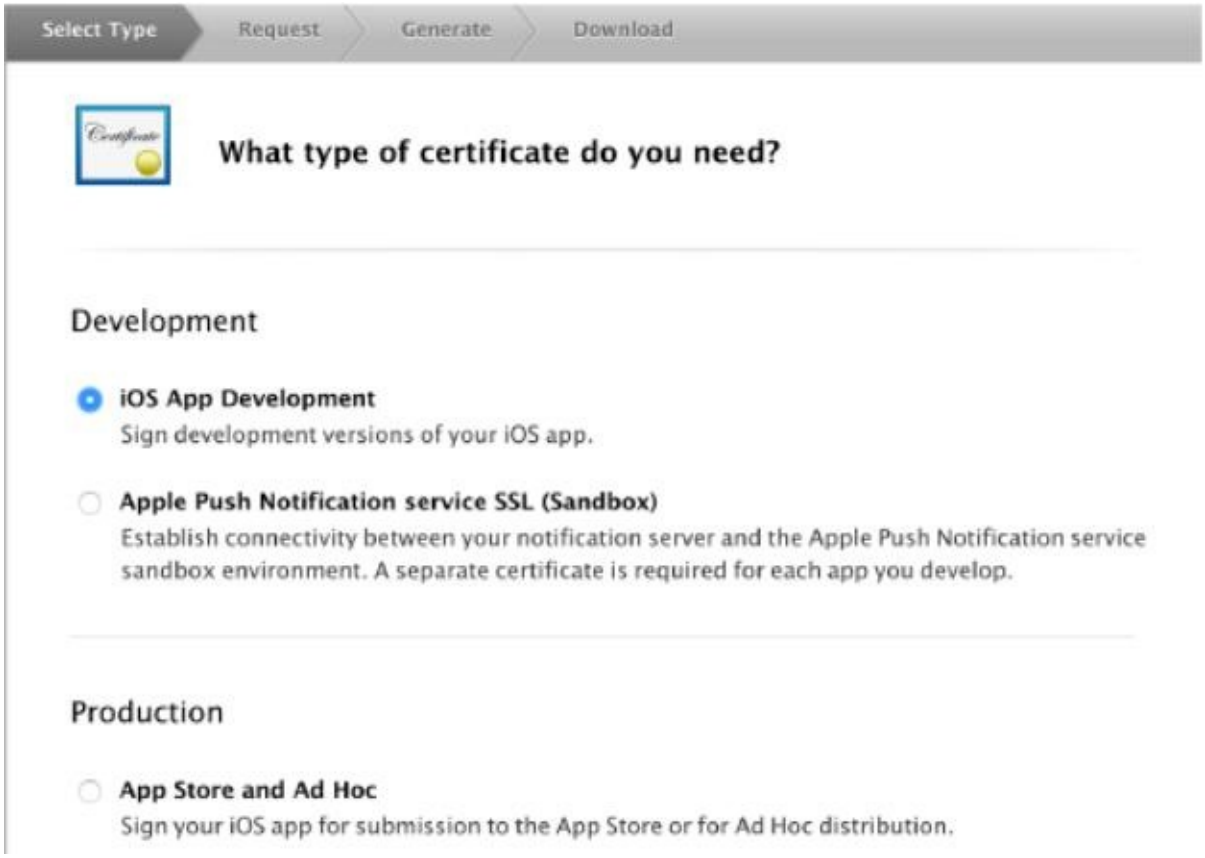
创建一个证书

要完成接下来的步骤需要登录到App Member Center然后去**Certificates,Idenfitiers & Profiles**部分：



- 选择**iOS App**下面的**Certificates**
- 点击右上的**+**按钮：

接下来选择要生成什么类型的证书。测试应用的话选择**iOS App Development**，如果是发布时话选择**App Store and Ad Hoc**。



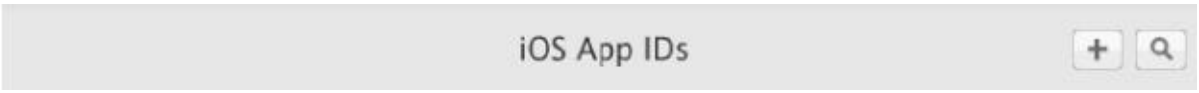
可能你注意到了屏幕上的其他选项。如果你想要使用类似Apple Push Notifications服务或者WatchKit的话，你需要另外创建证书，但是我们用不上。

- 选择证书类型然后点击**Continue**
- 现在就要你刚才通过Keychain Access创建的证书签名申请了，点击继续然后上传签名申请。一旦选择了**.certSigningRequest**文件点击**Generate**。

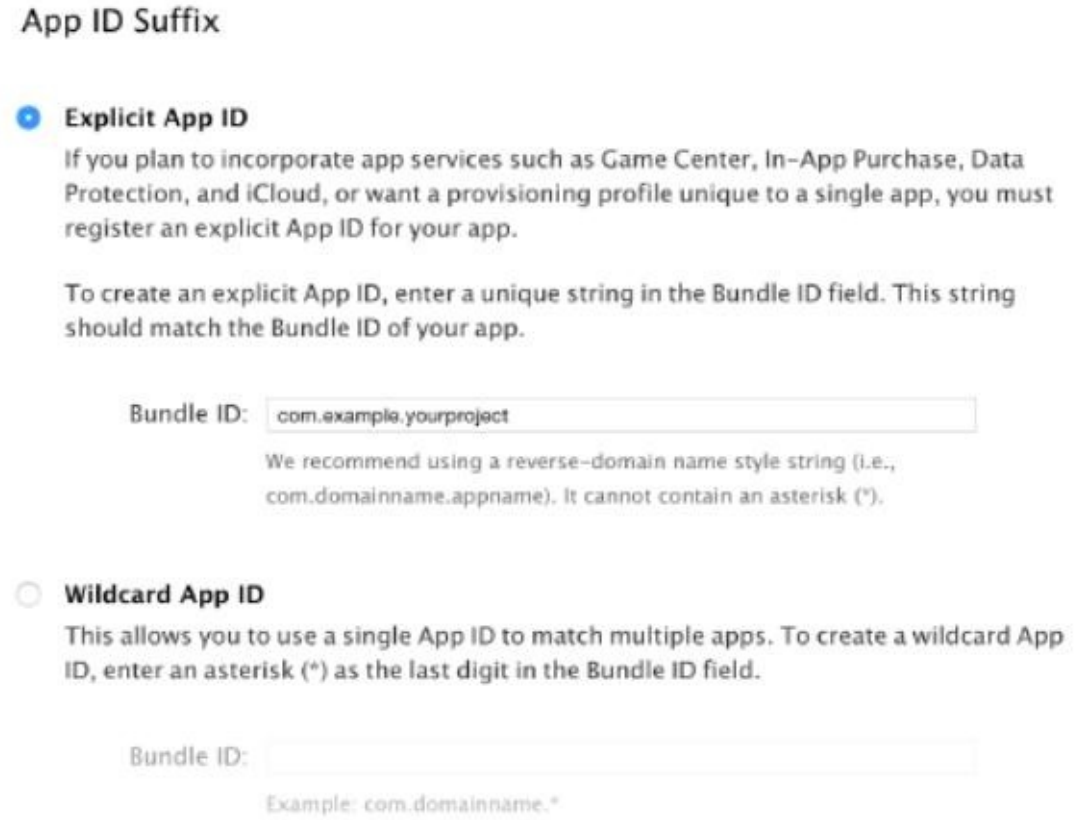
现在你可以下载你的证书了。将他下载到一个安全的地方，然后打开安装（方便起见，将他放到OpenSLL的bin文件夹内，例如 **OpenSSL-Wind64/bin**，因为这是我们运行命令的地方）。

制作**Identidfier**

- 点击**App IDs**然后的点击 **+** 图标



- 填写App ID Description然后提供一个Explicit App ID就像这样：



这个必须和**config.xml**里面的**id**一致。

- 来到屏幕底部点击**Continue**
- 下一个屏幕点击**Submit**

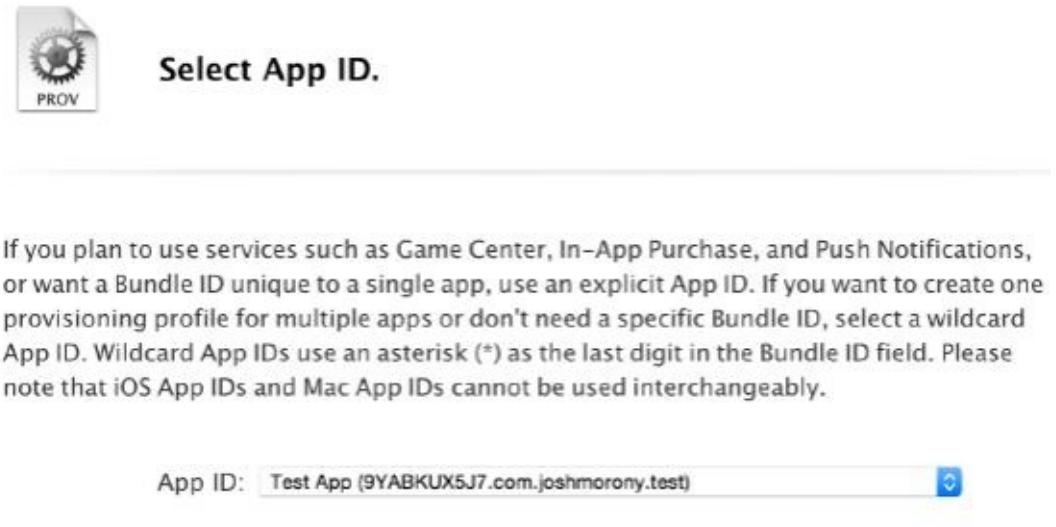
此时App ID注册完成，可以在配置文件中使用了。

创建一个配置文件（**Provisioning Profile**）

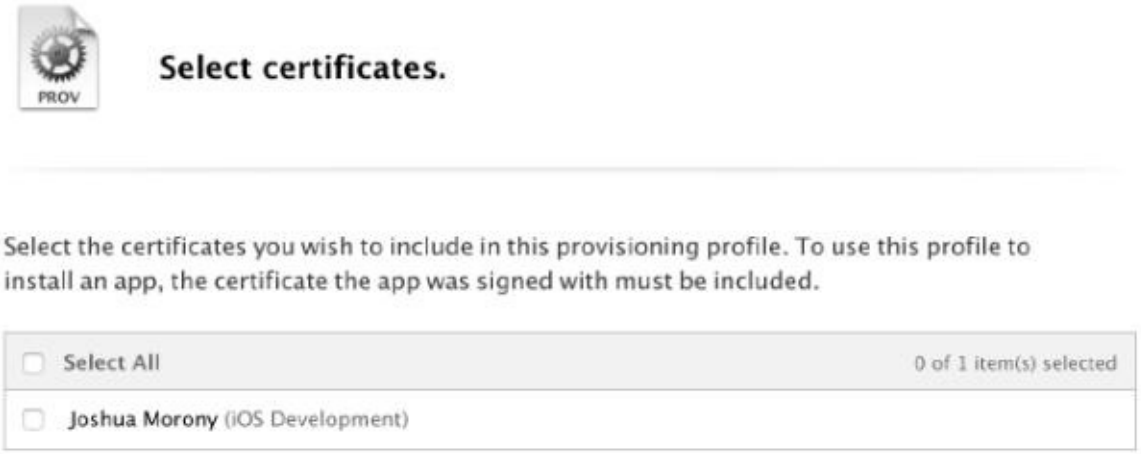
创建配置文件之前需要添加应用可以运行的设备。

- 点击**Devices**然后点有右上的 **+** 按钮

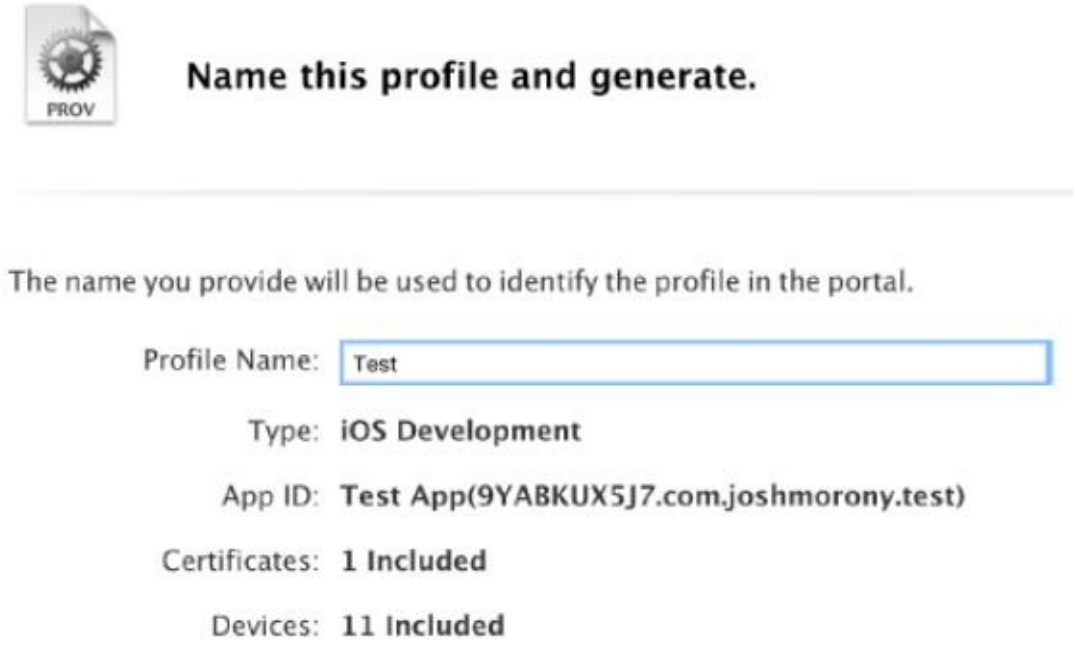
- 如果是给测试制作配置文件的话选择**iOS App Development**，如果是发布版的话那么选择**App Store**。点击**Continue**
- 选择刚才创建的App ID然后点击**Continue**



- 选择想要用的证书然后点击**Continue**



- 选择所有可以想要运行的设备然后点击**Continue**：
- 给配置文件提供一个名字然后点击**Generate**



现在你应该可以下载你的配置文件了。（当然，将他下载到OpenSSL的bin文件夹内这样接下来的步骤就会简单些）

生成 .p12

最后，我们将使用从Member Center下载的证书和一些命令来制作**.p12**。

- 运行如下命令生成一个PEM文件：

```
openssl x509 -in ios_development.cer -inform DER -out app_pem_file.pem -outform PE
M
```

- 使用开始生产的密钥和刚才生成的PEM来制作你的.p12文件：

```
openssl pkcs12 -export -inkey mykey.key -in app_pem_file.pem -out app_p12.p12
```

这里很多地方都可能发生错误也很容易混淆。所以如果你遇到问题的话我的建议是重新开始整个流程，慢慢的小心的确保你每个输入都正确。
恭喜！现在你有了**.p12**文件。由于你有了配置文件和**.p12**文件，如果你是使用PhoneGap Build的话，在上传的时候你就可以把这些文件附加上去了。完成之后你就可以生成一个签名的**.ipa**文件用来安装到设备上（在胎哪家了设备和正确的执行了所有步骤的情况下！）。
我们将会简单的讨论如何使用PhoneGap Build进行构建。

在Mac或者PC上为Android应用签名

之前说过，Android和iOS有点不一样（幸运的是简单一些）。iOS在开发和产品阶段都需要证书和配置文件。
Android呢，只需要你在测试设备上安装一个‘debug’应用而不要上面那个流程。你只需要在将应用发布到Google Play的时候才需要对应用签名。Android需要你创建一个‘keystore’来对应用进行签名。

对Android应用签名

对Android应用签名简单一些，我之前也说过，如果你只是测试的话你都不用对他进行前（只有在发布到Google Play的时候才需要签名）。给Android应用签名我们需要创建一个**keystor**文件。
给应用签名需要一个Android SDK自带的工具叫做**keytool**。如果你的电脑上没有按组航Android SDK的话，请参考这个指引：
http://ionicframework.com/docs/v1/ionic-cli-faq/#android-sdk
设置好Android SDK之后，可以按照如下步骤创建一个keystore文件。

- 运行如下命令生成一个指定别名**alias_name**（你应该改一下）的keystore文件

```
keytool -genkey -v -keystore my-release-key.keystore -alias alias_name -keyalg RSA
 -keysize 2048 -validity 10000
```

  输入这个命令的时候会提醒你输入密码 -- 输入个密码存好并记住。然后会问你一系列的问题，喜欢的话大部分问题你都可以不回答。最后，会问你要一个keystor密码。
  现在你有一个keystore文件了，可以用来给你的应用签名（我们会讨论他的工作原理）。
  重点：为了后面能在应用商店更新你的应用你需要这个keystore文件，以及别名和密码。如果你忘记了其中一个的话，就更新不了。

生成一个Key Hash

如果你使用Facebook功能的话，那么会要求你创建一个Facebook应用，这个在制作CamperChat应用的时候学习过（本书的专家套装里面才有），那么你就需要通过你的keystore文件生成一个Key Hash提供给Facebook for Android。
做这个的话很简单，只需要运行如下命令就可以了：

```
keytool -exportcert -alias alias_name -keystore my-release-key.keystore | openssl sha1
 -binary | openssl base64
```

确保用你的keystore别名替换**alias_name**和用你的keystore文件路径替换**my-release-key.keystore**。
一旦完成的话会在终端输出你的Key Hash，然后你就可以简单复制到你的Facebook应用的Android platform setting去。

使用PhoneGap构建程序构建iOS和Android（无MAC）

有两种方法将Cordova/PhoneGap合并到你的应用中，你可以通过命令行来使用本地安装版，或者使用PhoneGap Build云服务。
选择很简单：

- 如果你有一台Mac：在本地使用Cordova
- 如果你不是构建iOS应用：在本地使用Cordova
- 如果你没有Mac但是想要创建iOS应用：使用PhoneGap Build

PhoneGap Build服务允许你免费创建一个私有应用。你可以删掉应用重用这个特权来构建大量应用，但是如果你想同时在PhoneGap Build中管理大量应用的话那么买一个吧。
重点：除非你是第三种情况（没有**Mac**但是想要构建**iOS**应用），不是的话直接跳过本课就可以了
PhoneGap Build是一个伟大的服务，但是允许的话你还是使用Cordova本地安装版。
使用本地Cordova的优点有：

- 由于不用使用PhoneGap Build服务器和下载生成的 .ipa文件（Android的是.apk），所以会快很多
- 你不需要耗费数据去下载和上传文件，开发也不需要网络连接
- 你可以访问大量插件
- 你可以控制应用（你可以直接访问本地包装，而不是网络文件）
- PhoneGap Build要钱，如果你想要不止一个私有项目的话

我提过PhoneGap Build可以在没有Mac的情况下构建iOS应用。他也可以在没有Android SDK的情况下创建Android应用。这就是为什么他跟你当前使用的操作系统无关，因为所有的编译都是在PhoneGap Build的服务端进行的。
使用PhoneGap Build的时候，你可以上传你的应用代码（纯粹的基于互联网的代码，即：HTML，CSS和Javascript）然后就有为iOS和Android编译好的应用返回（Windows Phone的也可以）。大体步骤如下：

1. 构建你的应用
2. 创建一个config.xml告诉PhoneGap Build如何构建【译者：编译或者打包好些，但是作者都是用的build】你的应用【译者：应该是ionic编译生成的代码】
3. 打包应用【译者：ionic项目生成的js相关】
4. 上传到PhoneGap Build
5. 下载本地包结果

编译和SDK整合都是在他们的后端操作的，这个方法比在自己机器上设置好所有东西简单得多（但是长期使用的话麻烦些）。最重要的是他让没有**Mac**的人可以制作**iOS**应用。
目前为止你应该有一个完成的应用可以用于编译，我们来一步一步学习如何使用PhoneGap Build。

通过PhoneGap Build构建

上传到PhoneGap Build的时候我们只想包含构建好了的网页相关文件，也就是我们项目**www**文件夹里的所有内容。其他文件和文件夹主要是用户配置Cordova和构建流程的，这些是我们在使用PhoneGap Build的时候不需要用的。
在上传到PhoneGap Build之前我们还需要做一点改动。

制作一个config.xml文件

你可以已经知道了，项目里包含了一个**config.xml**用于配置项目如何使用Cordova构建。我们还是要用到这个文件，但是PhoneGap Build需要的有点不一样。
警告：接下来的代码量有点多，这个格式也很难读，如果喜欢的话你可以从下载包中的PhoneGap Build应用范例从拷贝这些代码。
> 创建文件**www/config.xml**添加如下代码：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- Change the id to something like com.yourname.yourproject -->
<widget xmlns = "http://www.w3.org/ns/widgets"
    xmlns:gap = "http://phonegap.com/ns/1.0"
    id = "com.example.project"
    versionCode = "10"
    version = "1.0" >
<!-- versionCode is optional and Android only -->
<name>App Name</name>
<description>
Description
</description>
<author href="http://www.example.com" email="you@yourdomain.com">
Your Name
</author>
<plugin name="cordova-sqlite-storage" source="npm" />
<plugin name="com.phonegap.plugin.statusbar" source="pgb" />
<plugin name="org.apache.cordova.splashscreen" source="pgb" />
<plugin name="com.indigoway.cordova.whitelist.whitelistplugin" source="pgb" />
<preference name="prerendered-icon" value="true" />
<preference name="target-device" value="universal" />
<preference name="android-windowSoftInputMode" value="stateAlwaysHidden"/>
<icon src="resources/android/icon/drawable-ldpi-icon.png" gap:platform="android" gap:q
ualifier="ldpi"/>
<icon src="resources/android/icon/drawable-mdpi-icon.png" gap:platform="android" gap:q
ualifier="mdpi"/>
<icon src="resources/android/icon/drawable-hdpi-icon.png" gap:platform="android" gap:q
```

```
ualifier="hdpi"/>
<icon src="resources/android/icon/drawable-xhdpi-icon.png" gap:platform="android" gap:
qualifier="xhdpi"/>
<icon src="resources/android/icon/drawable-xxhdpi-icon.png" gap:platform="android" gap
:qualifier="xxhdpi"/>
<icon src="resources/android/icon/drawable-xxxhdpi-icon.png" gap:platform="android" ga
p:qualifier="xxxhdpi"/>
<icon src="resources/ios/icon/icon.png" gap:platform="ios" width="57" height="57"/>
<icon src="resources/ios/icon/icon@2x.png" gap:platform="ios" width="114" height="114"
/>
<icon src="resources/ios/icon/icon-40.png" gap:platform="ios" width="40" height="40"/>
<icon src="resources/ios/icon/icon-40@2x.png" gap:platform="ios" width="80" height="80"
/>
<icon src="resources/ios/icon/icon-50.png" gap:platform="ios" width="50" height="50"/>
<icon src="resources/ios/icon/icon-50@2x.png" gap:platform="ios" width="100" height="1
00"/>
<icon src="resources/ios/icon/icon-60.png" gap:platform="ios" width="60" height="60"/>
<icon src="resources/ios/icon/icon-60@2x.png" gap:platform="ios" width="120" height="1
20"/>
<icon src="resources/ios/icon/icon-60@3x.png" gap:platform="ios" width="180" height="1
80"/>
<icon src="resources/ios/icon/icon-72.png" gap:platform="ios" width="72" height="72"/>
<icon src="resources/ios/icon/icon-72@2x.png" gap:platform="ios" width="144" height="1
44"/>
<icon src="resources/ios/icon/icon-76.png" gap:platform="ios" width="76" height="76"/>
<icon src="resources/ios/icon/icon-76@2x.png" gap:platform="ios" width="152" height="1
52"/>
<icon src="resources/ios/icon/icon-small.png" gap:platform="ios" width="29" height="29"
/>
<icon src="resources/ios/icon/icon-small@2x.png" gap:platform="ios" width="58" height=
"58"/>
<gap:splash src="resources/android/splash/drawable-land-ldpi-screen.png" gap:platform=
"android" gap:qualifier="land-ldpi"/>
<gap:splash src="resources/android/splash/drawable-land-mdpi-screen.png" gap:platform=
"android" gap:qualifier="land-mdpi"/>
<gap:splash src="resources/android/splash/drawable-land-hdpi-screen.png" gap:platform=
"android" gap:qualifier="land-hdpi"/>
<gap:splash src="resources/android/splash/drawable-land-xhdpi-screen.png" gap:platform=
"android" gap:qualifier="land-xhdpi"/>
<gap:splash src="resources/android/splash/drawable-land-xxhdpi-screen.png" gap:platform
="android" gap:qualifier="land-xxhdpi"/>
<gap:splash
src="resources/android/splash/drawable-land-xxxhdpi-screen.png" gap:platform="android"
gap:qualifier="land-xxxhdpi"/>
<gap:splash src="resources/android/splash/drawable-port-ldpi-screen.png" gap:platform=
"android" gap:qualifier="port-ldpi"/>
<gap:splash src="resources/android/splash/drawable-port-mdpi-screen.png" gap:platform=
"android" gap:qualifier="port-mdpi"/>
<gap:splash src="resources/android/splash/drawable-port-hdpi-screen.png" gap:platform=
"android" gap:qualifier="port-hdpi"/>
<gap:splash src="resources/android/splash/drawable-port-xhdpi-screen.png" gap:platform=
"android" gap:qualifier="port-xhdpi"/>
<gap:splash src="resources/android/splash/drawable-port-xxhdpi-screen.png" gap:platform
```

```
="android" gap:qualifier="port-xxhdpi"/>
<gap:splash
src="resources/android/splash/drawable-port-xxxhdpi-screen.png" gap:platform="android"
gap:qualifier="port-xxxhdpi"/>
<gap:splash src="resources/ios/splash/Default-568h@2x~iphone.png" gap:platform="ios" w
idth="640" height="1136"/>
<gap:splash src="resources/ios/splash/Default-667h.png" width="750" gap:platform="ios"
height="1334"/>
<gap:splash src="resources/ios/splash/Default-736h.png" width="1242" gap:platform="ios"
 height="2208"/>
<gap:splash src="resources/ios/splash/Default-Landscape-736h.png" gap:platform="ios" w
idth="2208" height="1242"/>
<gap:splash src="resources/ios/splash/Default-Landscape@2x~ipad.png" gap:platform="ios"
 width="2048" height="1536"/>
<gap:splash src="resources/ios/splash/Default-Landscape~ipad.png" gap:platform="ios" w
idth="1024" height="768"/>
<gap:splash src="resources/ios/splash/Default-Portrait@2x~ipad.png" gap:platform="ios"
width="1536" height="2048"/>
<gap:splash src="resources/ios/splash/Default-Portrait~ipad.png" gap:platform="ios" wi
dth="768" height="1024"/>
<gap:splash src="resources/ios/splash/Default@2x~iphone.png" gap:platform="ios" width=
"640" height="960"/>
<gap:splash src="resources/ios/splash/Default~iphone.png" gap:platform="ios" width="32
0" height="480"/>
<!-- Default Icon and Splash -->
<icon src="resources/icon.png" />
<gap:splash src="resources/splash.png" />
<access origin="*"/>
</widget>
```

这个文件是针对Quicklists应用的 -- 注意我们是如何通过几行代码包含插件的？PhoneGap Build纳入插件的方式和Cordova不一样。你可能记得我们在制作每个应用的开始会运行一些这样的命令：

```
ionic plugin add [plugin name]
```

来设置插件。这会在项目中本地设置好插件，但是在PhoneGap Build中使用插件我们需要在新的**config.xml**中这么指定：

```
<plugin name="cordova-sqlite-storage" source="npm" />
<plugin name="com.phonegap.plugin.statusbar" source="pgb" />
<plugin name="org.apache.cordova.splashscreen" source="pgb" />
<plugin name="com.indigoway.cordova.whitelist.whitelistplugin" source="pgb" />
```

虽然大部分情况下，不是所有插件都在PhoneGap Build中可用。当给项目添加插件的时候，只要搜索哟一些PhoneGap Build安装须知，大部分插件都会告诉我们如何加入PhoneGap Build。如果你用PhoneGap Build构建本书的任何范例，应该先看一下准备工作部分然后确保

在**config.xml**中添加好所有插件。
同时记得将顶部的**id**替换为你自己的Bundle ID，即： com.yourproject.yourname。

复制资源

另一个主要不同是包含了资源（图标和闪屏）。我们也要去**config.xml**文件里添加这些引用，也要将他们复制进来，因为他们现在在**www**文件夹外面（这样的话在上传的时候不会上传上去）。
将**resources**文件夹拷贝到**www**里。
在拷贝资源的同时，也需要在资源文件夹内创建一个空白文件名为*.pgbomit*。这是告诉PhoneGap Build这个文件内的文件只会用于闪屏和图标，在应用内是不能使用的（会占用很大的空间）。

上传到PhoneGap Build

现在万事俱备，我们只需要将**www**里面的内容（不是**www**文件夹本身）打包成zip然后上传到PhoneGap Build。如果你还没有帐号的，先注册一个吧https://build.phonegap.com/
有了账号之后直接创建一个新的app然后上传刚才创建的*.zip*文件点击“Ready to Build”。iOS构建开始会失败，Android会成功。这是因为你不需要在开发阶段对Android应用签名，但是iOS的要。
当你构建发行版的时候，或者想要下载iOS版本，你需要上传上一章制作的签名密钥。意思就是你需要想iOS构建提供**.p12**文件和配置文件，给Android提供**keystore**。
应用完成构建的时候，就可以下载iOS的**.ipa**和Android的**.apk**文件。我们稍后讨论如何将它们上架到应用商店。

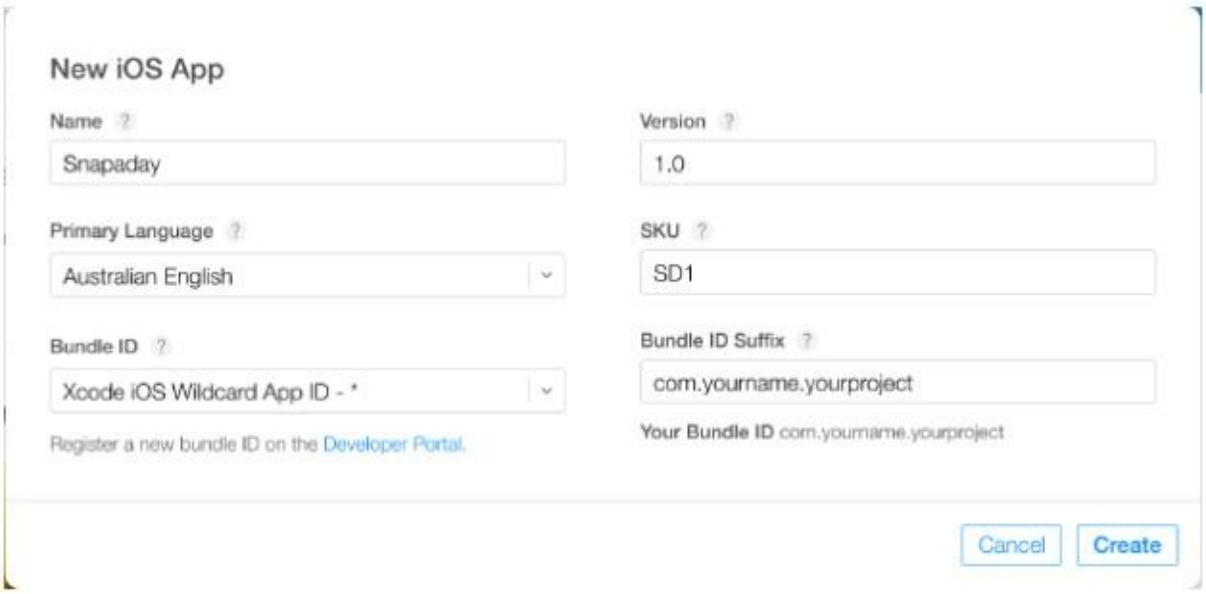
提交到Apple App Store

将我们的应用扔到apple脸上的时候终于来临了！在此之前，请一定务必要数一下 App Store Review Guideline -- 如果不遵从这些条款的话，你的应用会被拒。
注意：如果没有制作好你自己的应用的话，不要尝试下面的步骤 -- 你不能上传本书的任何范例
想要上传应用到App Store的话你需要创建一个App Store Listing，当然还有上传应用。

创建App Store Listing

我们先从创建App Store Listing开始。需要做的事情很多，但是都很简单。

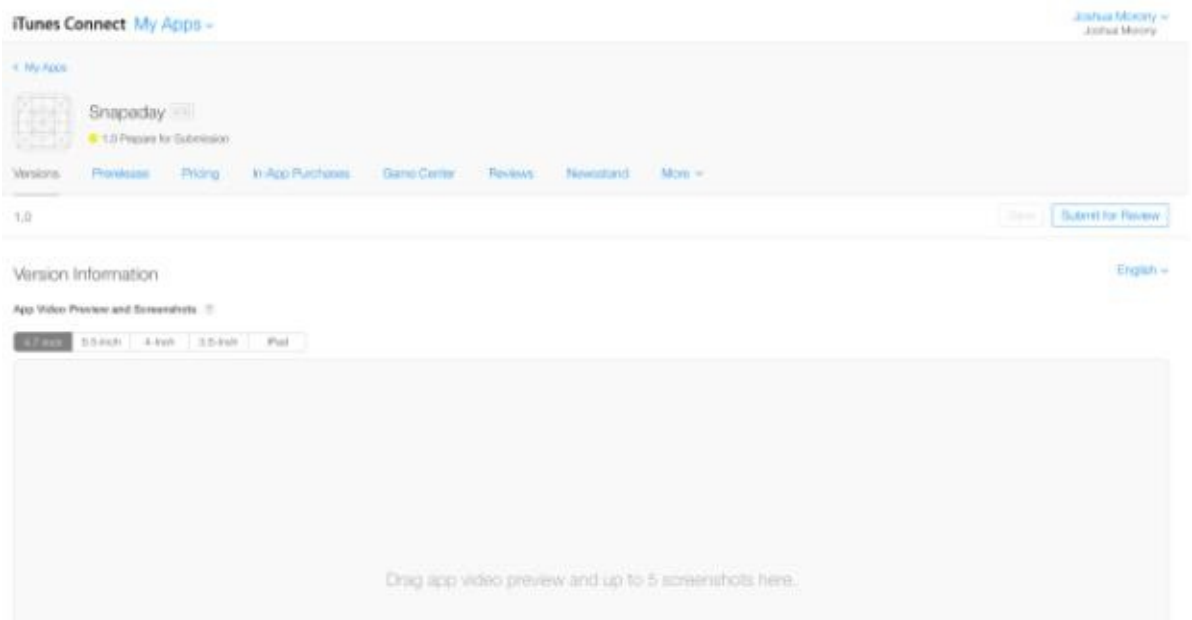
- 登录iTunes Connect
- 去到**My Apps**
- 选择左边的 + 图标然后选择**New iOS App**
- 根据提示填写信息然后点击**Create**：



如果你是用XCode提交应用的话你可以使用**XCode iOS Wildcard App ID**，或者你可以选择在iOS认证课程中创建的Bundle ID。SKU不需要什么特别的事情，只是给你参考而已并且**Bundle ID Suffix**必须和你在**config.xml**文件中的**id**一致。

现在应该可以看到这样一个仪表盘：

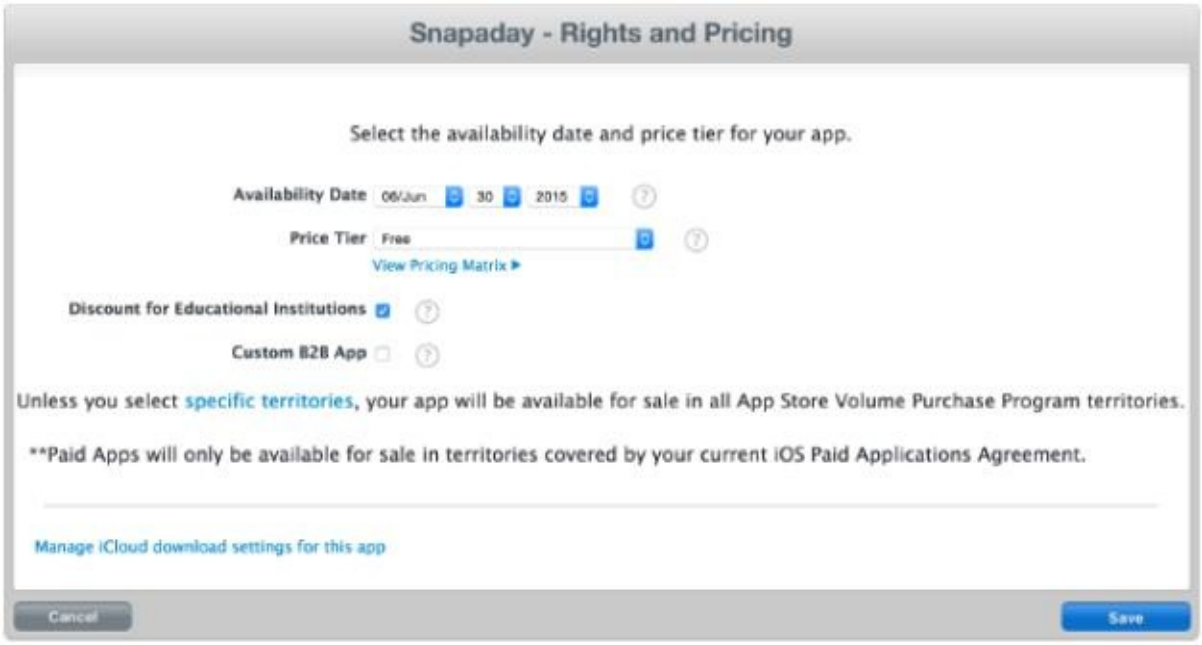
- 在iTunes Connect中打开你的新应用，应该可以看到这样的仪表盘：



- 填好本页的所以信息（包括不同尺寸设备的大量截屏）

注意：本页你应该可以看到一个Build部分。在上传好应用之后，会来到这个部分（下一部分会讲）然后选择你需要上传的构建版。

- 点击**Pricing**标签页，填写如下信息：



如果你想发布一个付费应用的话你需要接受一些额外的iTunes Connect条款。

上传应用

上传应用的途径不多，主要取决于你的应用格式和你使用的操作系统。想要发布iOS应用到应用商店的话你需要使用发布证书来给你的应用签名，再一次，实现这个的方法缺角与你使用的方法。
一旦上传应用后你就可以在iTunes Connect的**Build**部分看到他且可以附加到你的app store listing。

通过**XCode**提交应用

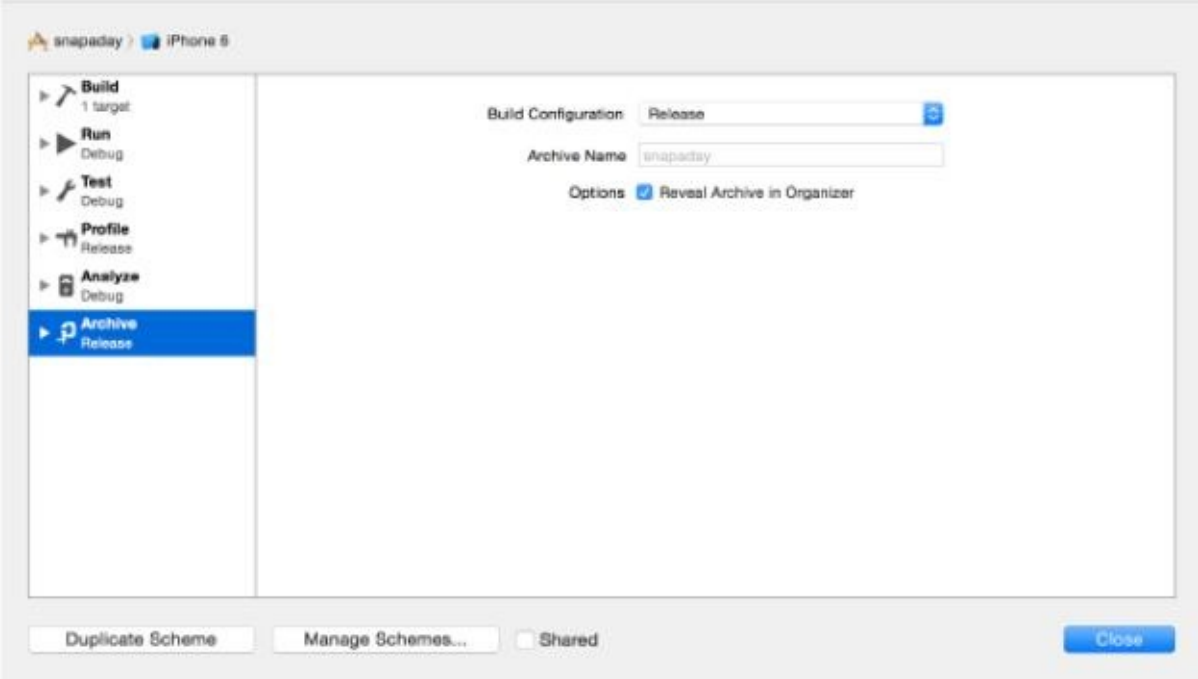
如果你有一台Mac的话你可以使用XCode来提交应用，这是一个很简单的途径。如果你没有Mac的话，而是通过PhoneGap Build生成了 *.ipa* 的话，那么你可以跳过使用Application Loader提交应用这一部分了。
继续之前需要在项目内运行：

```
ionic build ios
```

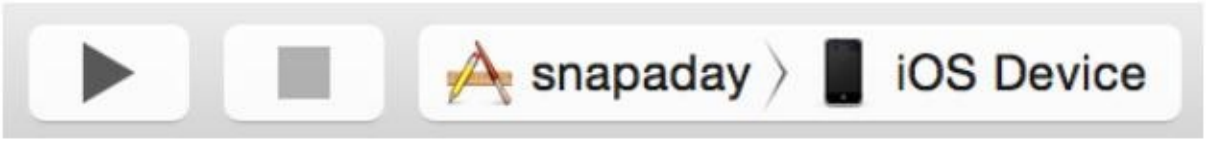
转换 **.xcodeproj** 文件到 **.xcarchive**

如果你有一个**.xcodeproj**文件（运行build命令的时候会生成）的话，那么你需要首先从他生成一个**.xcarhchive**文件。按照以下步骤生成即可：

- 双击 **.xcodeproj** 文件（位于 **platforms/ios/snapaday.xcodeproj**）就可以在XCode中打开他了
- 去往 **Product > Scheme > Edit Scheme**确保结构设置为**Release**配置：

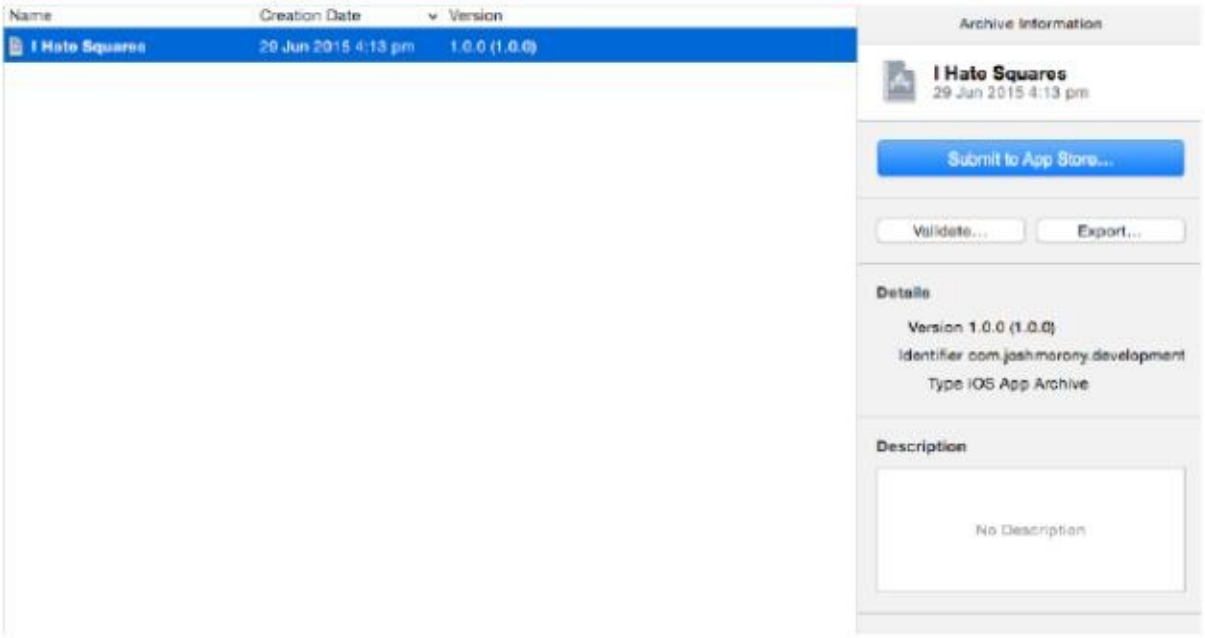


- 先确保在顶部有选中**iOS Device**或者**Generic iOS Device**，而不是模拟器：

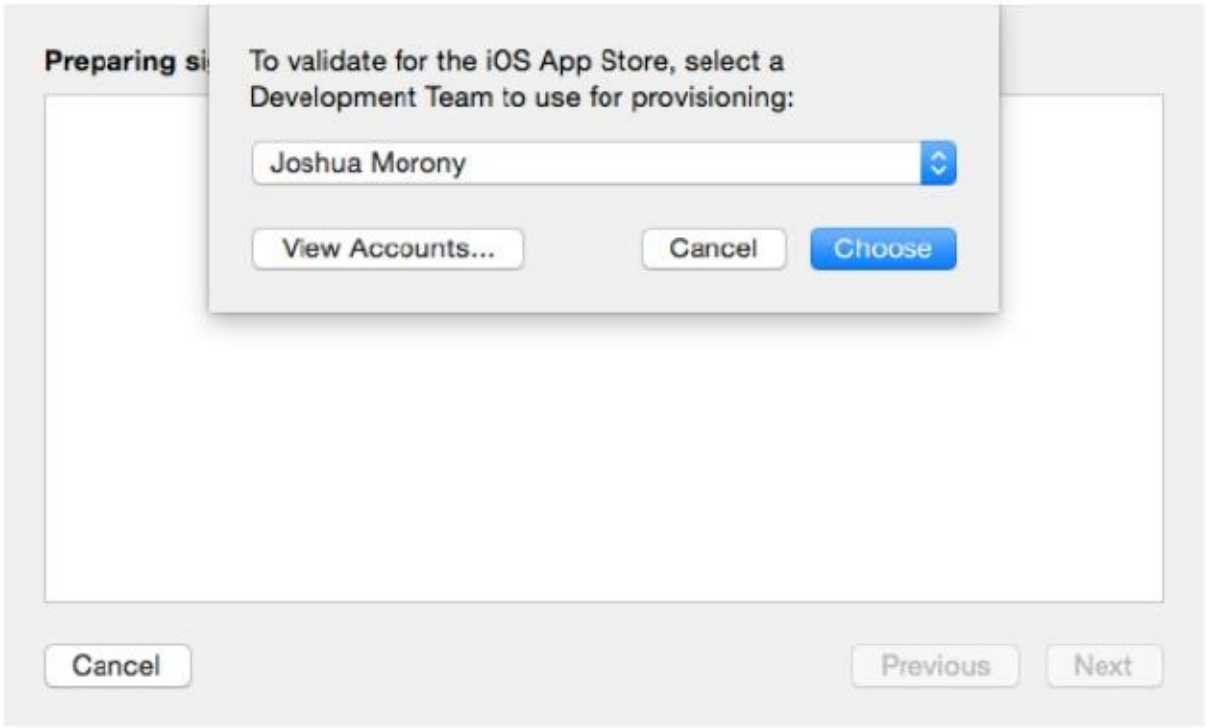


- 选择 **Product > Archive** 上传一个 **.xcarchive**文件。

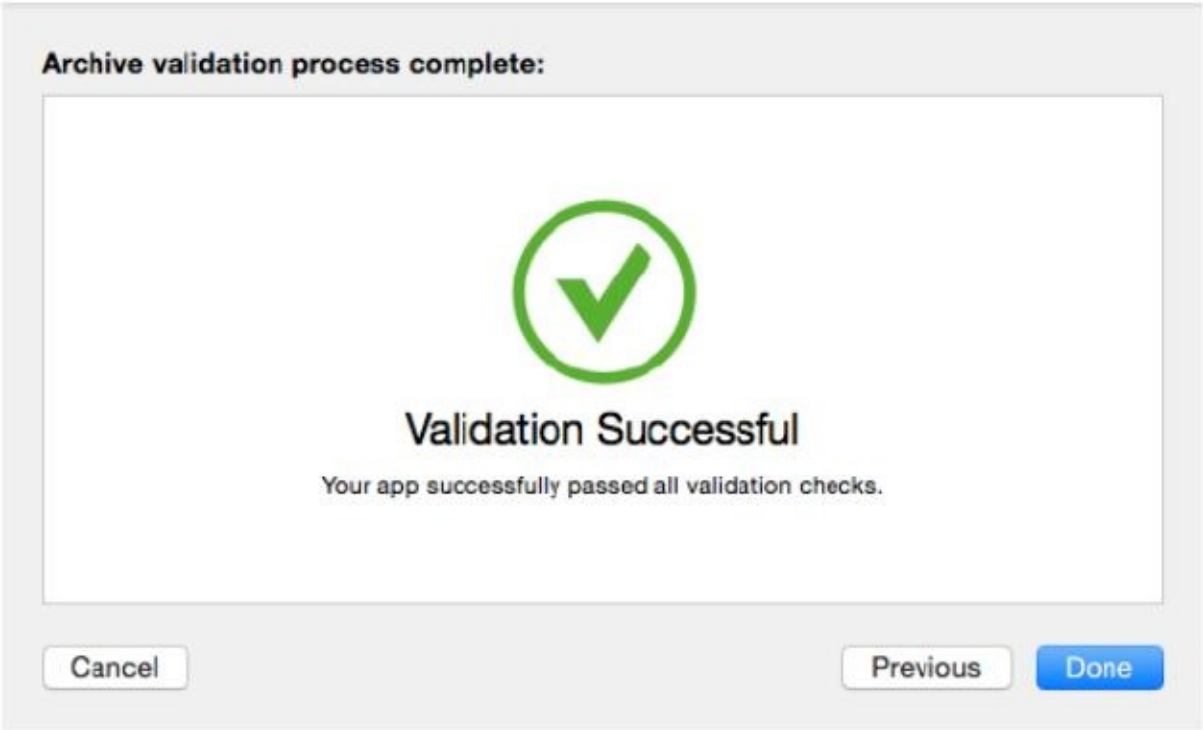
如果你上提交一个**.xcarchive**首先双击它在XCode中打开一下的屏幕（这个屏幕在你**Archive**你的应用的时候也会自动打开）：



首先你的选择你的存档然后点击**Validate..**按钮以确保所有设置正确。现在应该给你选择账户了：



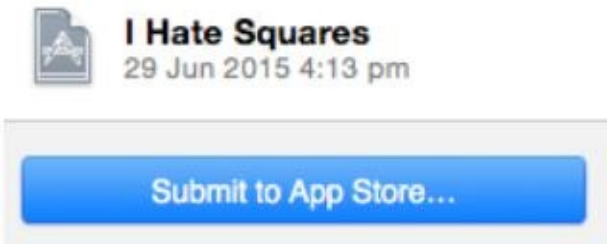
然后就可以看到你的应用显示出来了。点击**Validate**如果一切正常的话应该可以看到下面的提示：



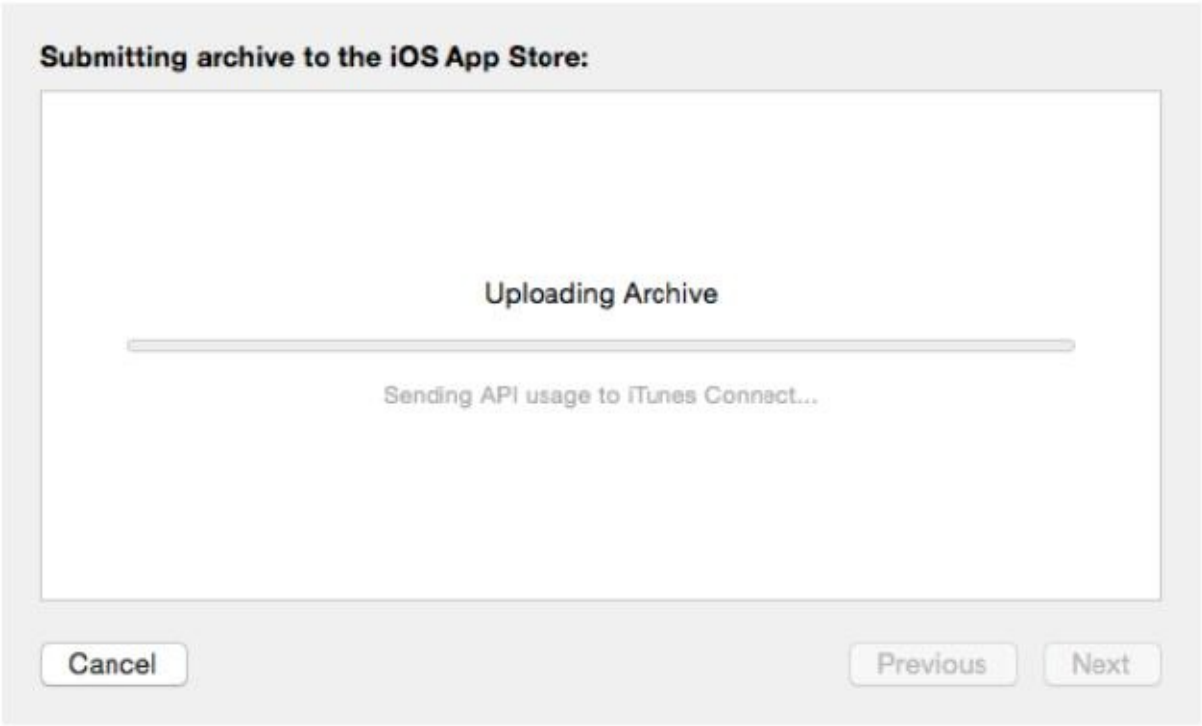
如果验证不成功，确保如下几点：

- 在iTunes Connect中设置好了应用
- 你完全遵照了iOS Certificates课程的指引
- config.xml里面的id和iTunes Connect里的Bundle ID Suffix是一样的

一旦成功验证你的项目，点击**Done**然后选择**Submit to App Store...**或者**Update to App Store...**：



现在你要跑一遍相同的流程，除了这次你选择**Submit**。点击**Submit**之后应用将开始上传到iTunes Connect：



使用**Application Loader**提交应用

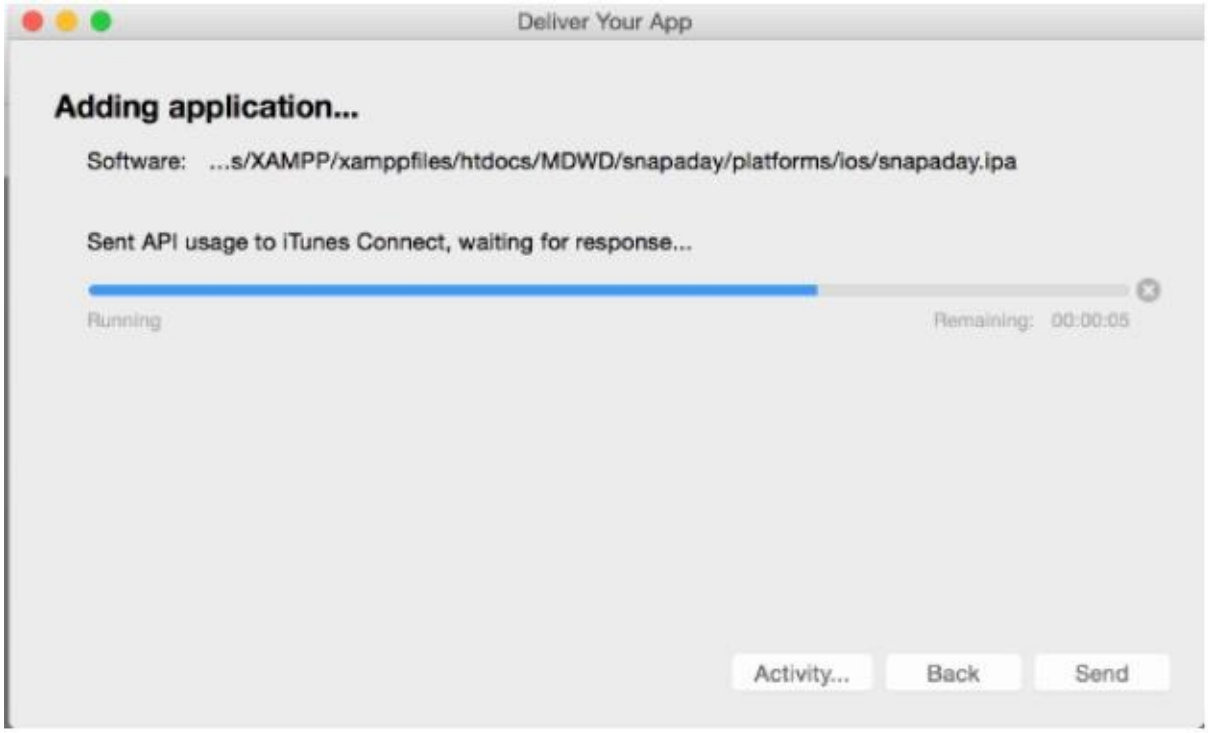
如果你没有Mac的话，那么你提交应用的唯一做法是提交构建和签名好了的**.ipa**。记住，如果是提交到应用商店的话，**.ipa**文件需要用发行证书签名而不是开发证书。
如果你还没有一个已经签名的**.ipa**文件的话，那么请先阅读**PhoneGap Build**课程。
你可以使用一个程序叫做**Appication Loader**提交**.ipa**到iTunes Connect，但不幸的是这个程序只有Mac版。理论上你在Windows上制作iOS应用只能上传上去，悲剧啊。
当然，还是可以想象其他的着，我个人喜欢这两个：

- 找朋友借Mac。只需要5分钟，所以如果你知道哪个朋友有Mac的话，把**.ipa**拷到一个USB盘，去朋友的Mac上下载Application Loader然后上传你的应用。
- Macincloud.com允许你远程登录一台Mac。这个服务是要钱的，你只有预付一些相当便宜的服务你就可以使用几分钟但是购买的有效期持续好多年（这就是我没有Mac之前干的）。

解决了访问Application Loader的问题之后，打开他，登录到你的iOS Developer 账户然后选择 Deliver Your App：

- 上传使用发布证书签名的 .ipa 文件然后点击Next

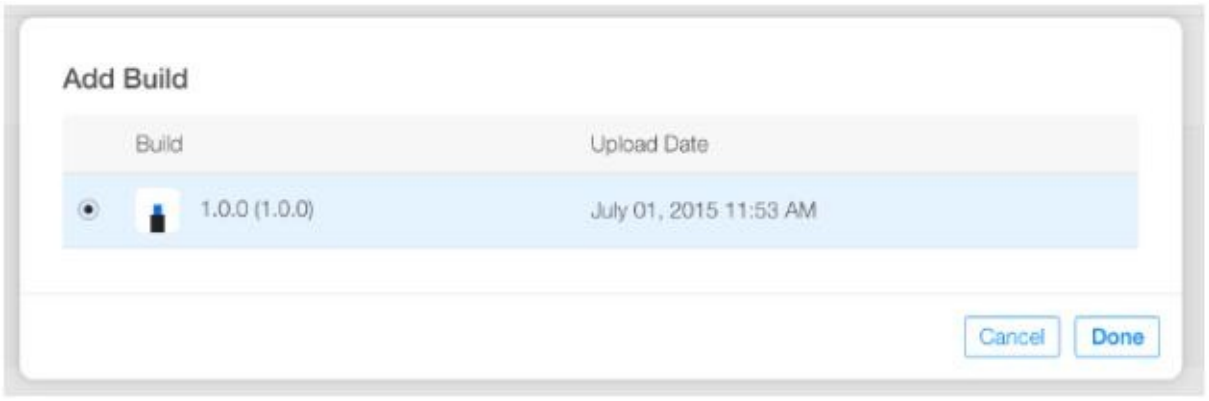
应用现状就上传到iTunes Connect了：



提交审核

上传完成后，就需要到iTunes Connect中完成你的app store listing。回到iTunes你的应用中然后去到**Build**部分：

这时候你会看到上面显示了一个 **+** 按钮。点击他，选择你刚才上传的构建版然后点击**Done**。



仔细检查列表里的东西，然后返回页面顶部，点击**Save**然后**Submit for Review**：

就可以提交应用到Apple了。现在，十指交叉耐心等待吧！Apple审核一般需要5-10天。虽然时间很长，但是你对他也是无能为力，除了耐心等待，别无他法。只要确保你严格遵循了Apple的规则和指引这样你的应用就不会被拒（否则的话你要进行修改重新提交并再次等待5-10天！）。

提交到Google Play

在经历了想Apple App store提交签名应用的噩梦之后，那么现在可以治疗一下了。相比之下，向Google Play提交应用就超级简单了。在开始之前，麻烦先注册Google Play Developer。
记住，提交应用之前需要用一个keystore文件对应用进行签名。
重点：如果你用了Crosswalk插件，那么在构建的时候会生成两个.apk。提交流程基本还是一样的，但是确保阅读本课最后的关于上传来两个.apk到同一个应用的注意事项。

打包Android应用

跟iOS应用不一样，无论是Mac还是Windows，打包Android应用都是同一个方法。如果你用的是PC，想要同时也打包iOS应用，那么意味着你要用PhoneGap Build了，由于你已经用过PhoneGap Build了，那么用于Android也没啥难度了。所以如果没有Mac的话我建议你用PhoneGap Build来打包Android，除非你只想打包Android。
如果你在使用PhoneGap Build的话，你已经有一了个签名了的**.apk**，这样你可以跳到下一步提交应用到**Google Play**。如果没有的话，按照如下步骤执行。
**> 创建文件 platforms/android/release-signing.properties 文件，添加如下内容：

```
storeFile=snapaday-release.keystore
keyAlias=snapaday
```

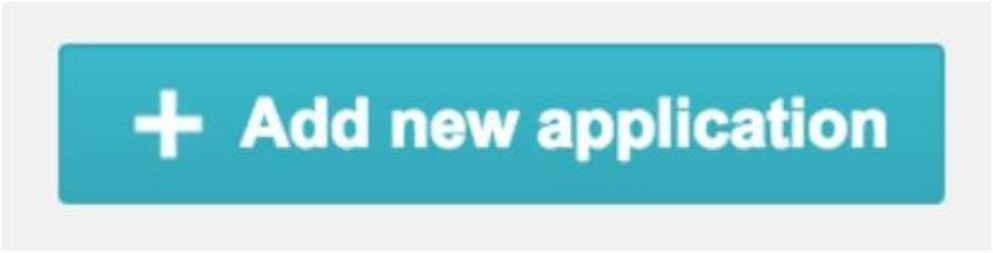
这个文件告诉打包流程如何对应用签名。这里需要提供在签名课程里面生成的**keystore**文件，还有keystore的别名。第一行应该是keystore文件的存放路径，为简单化我将keystore文件移动到了这个文件的目录下，当然你也可以指定一个喜欢的路径【译者：老外好罗嗦啊，有点受不了】。第二行是别名。
此文件就位之后，只要运行如下命令即可：

```
ionic build android --release
```

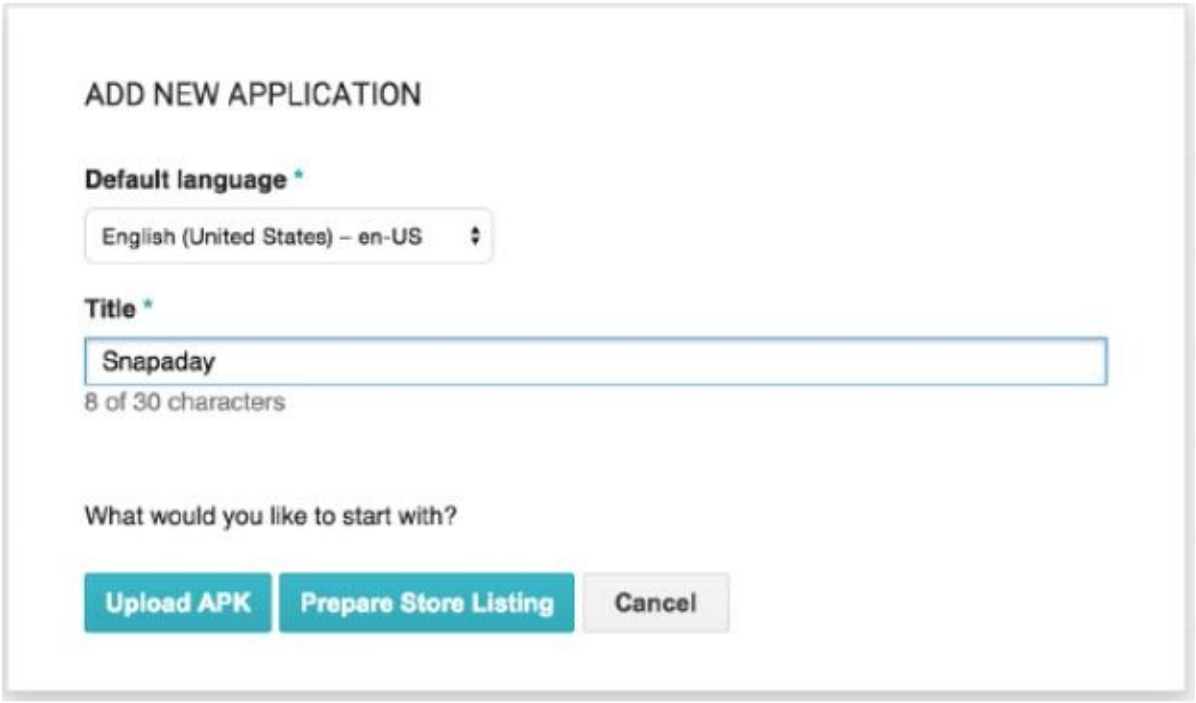
这个命令会给你生成**.apk**文件（如果使用了Crosswalk的话，不止一个了），位置在：**platforms/android/build/outputs/apk/**

提交应用到Google Play

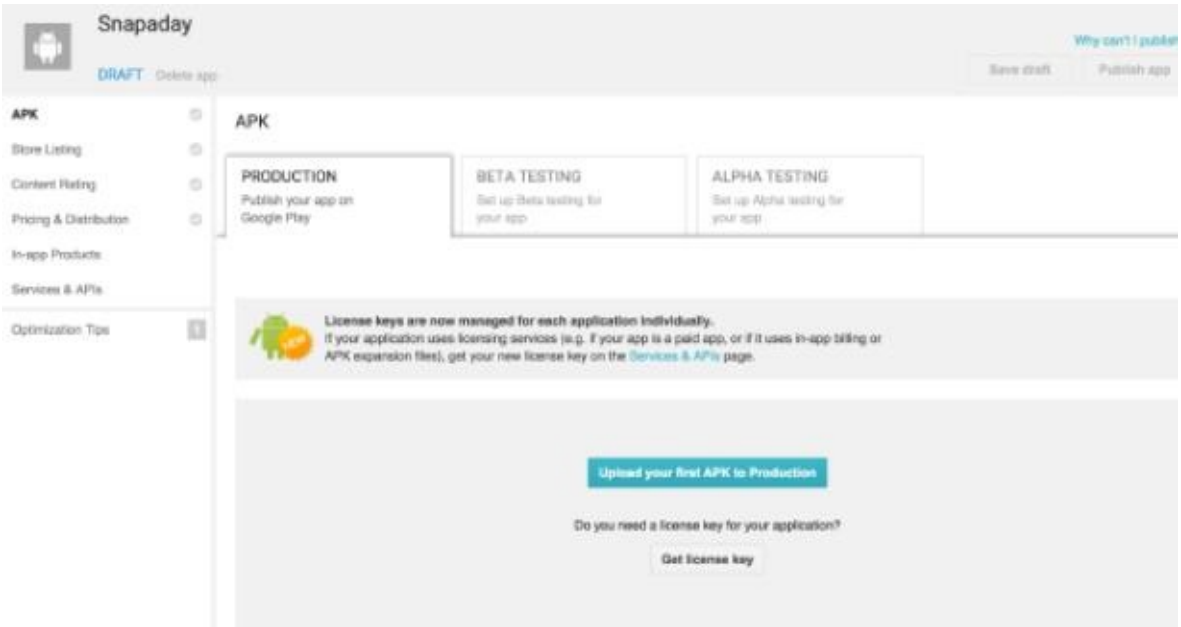
- 登录到Google Play Developer Console
- 点击**+Add New Application**:



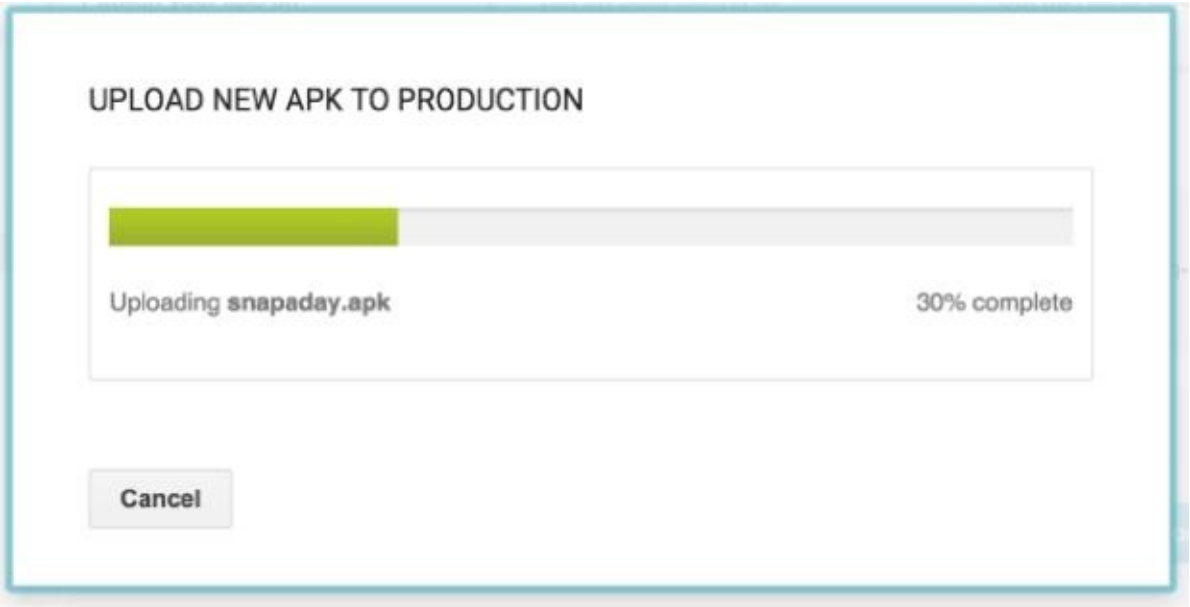
- 填好提醒框点击**Upload APK**



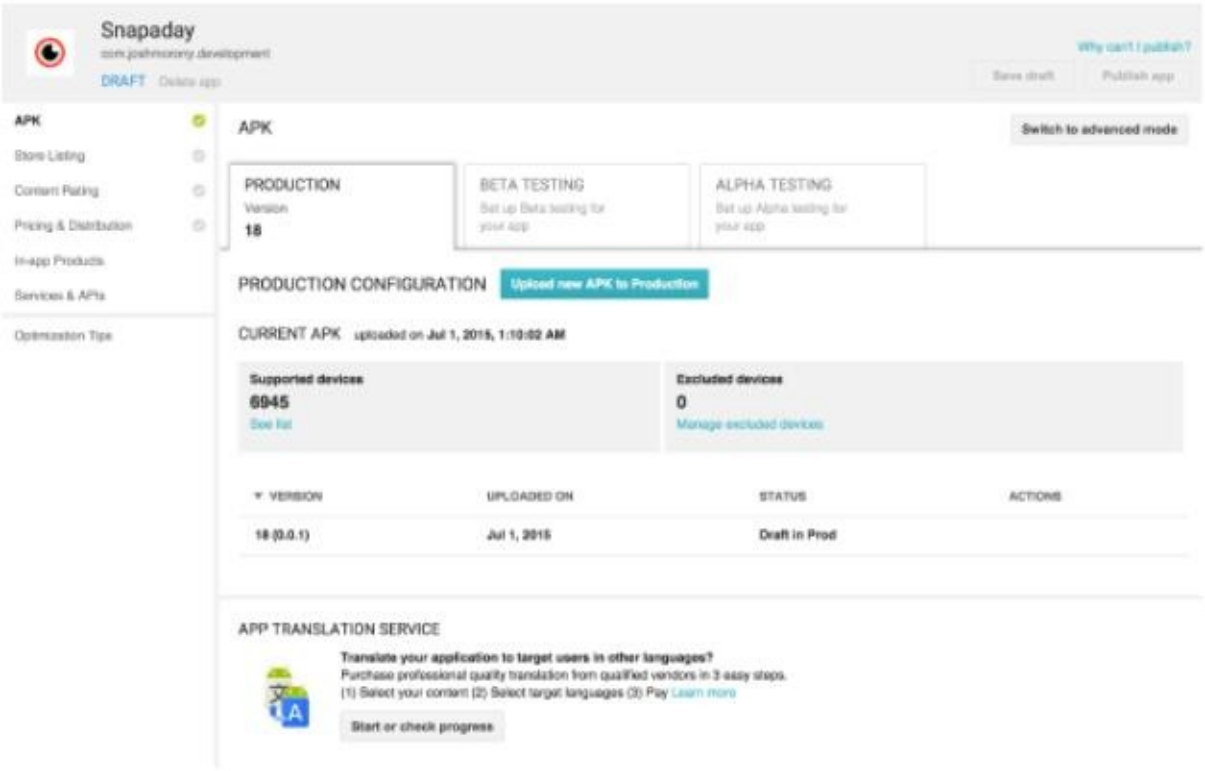
- 然后会看到这样的页面：



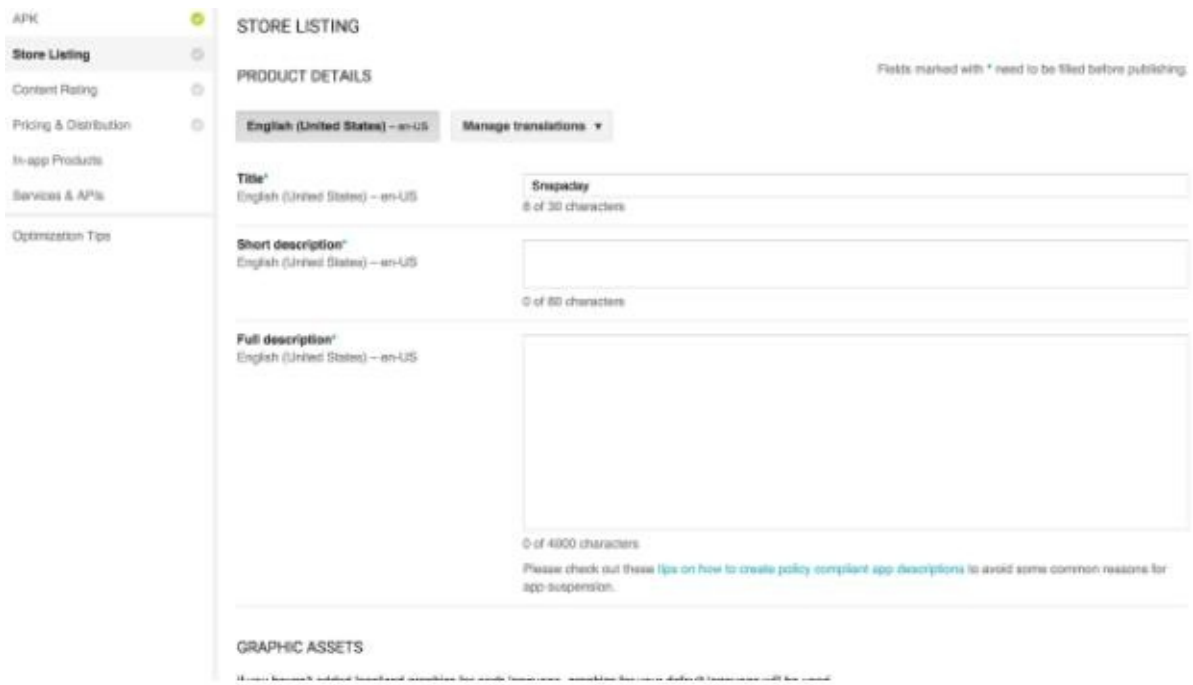
- 点击**Upload your first APK to Production**并上传签好名的**.apk**文件



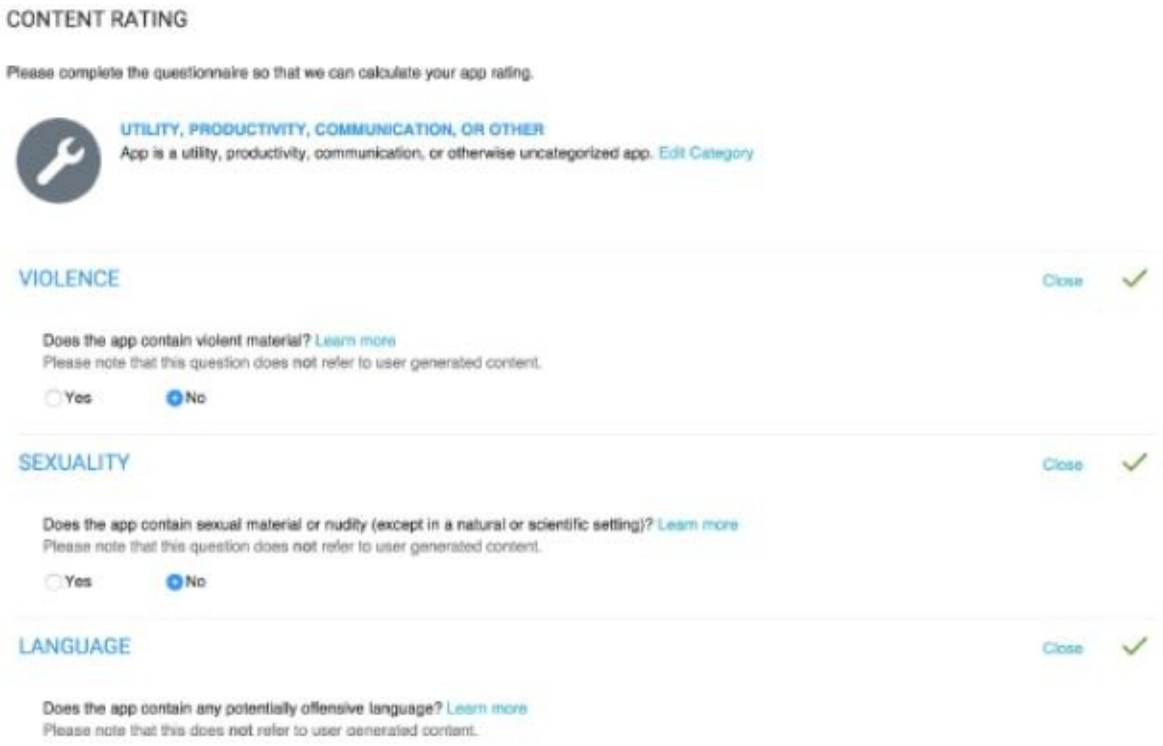
- 现在你可以看到页面更新：



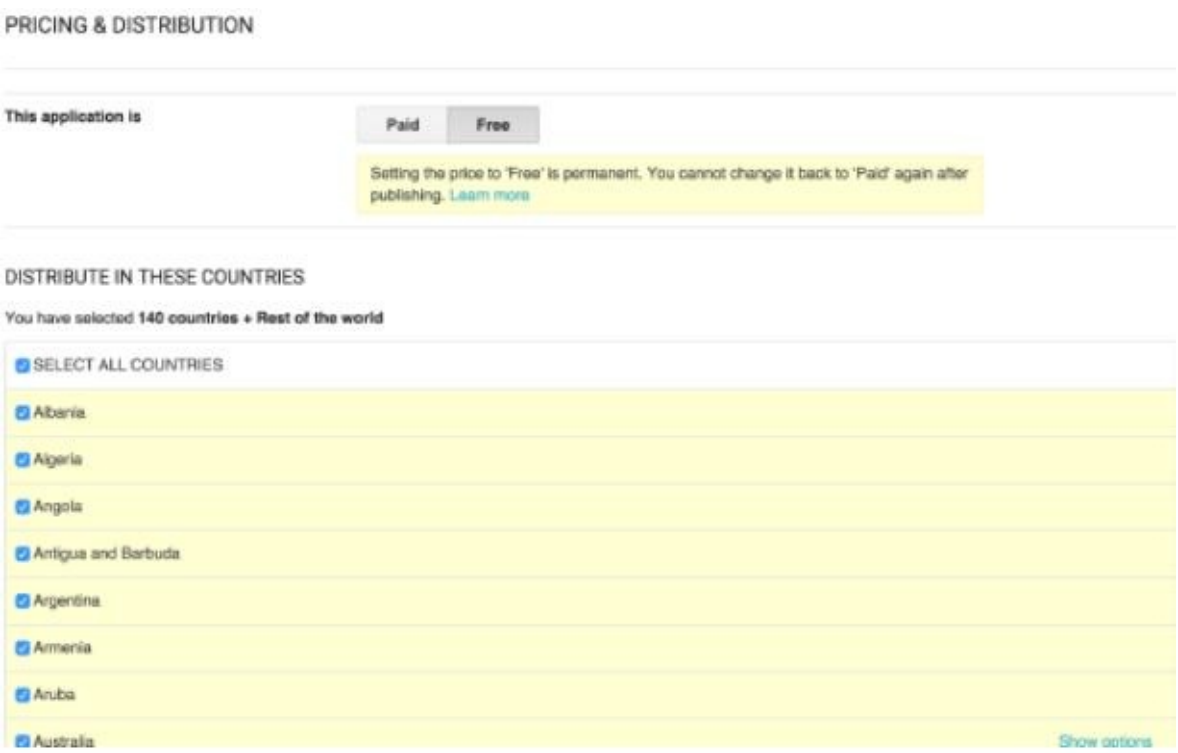
- 点击**Store Listing**填入信息包括截屏和促销素材然后点击**Save Draft**



- 接下来无**Content Rating**给应用创建一个内容评级点击**Save Draft**



- 去**Priceing & Distribution**填好信息点击**Save Draft**



完成上面这些页面的填写之后点击**Publish App**应用就提交上去了！
跟Apple App Store不同，你的应用数小时内就可以在Google Play上看到了。

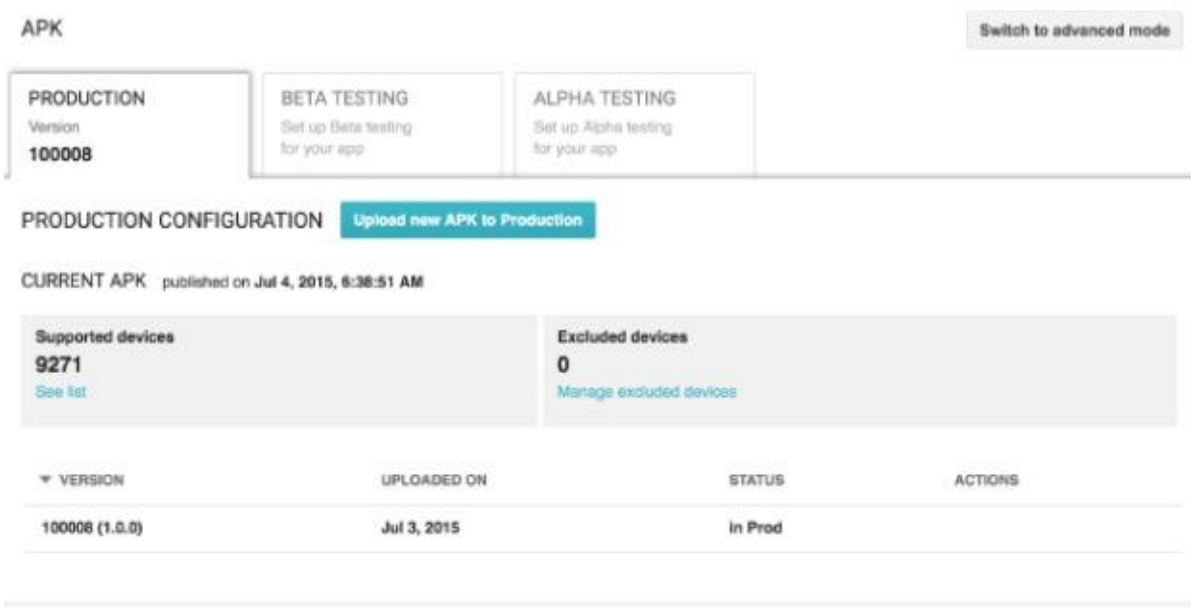
上传**Crosswalk**的多个**APK**

crosswalk的有趣之处在于会生成两个.apk文件，每个用户根据设备的不同需要的不同的.apk。现在在**platforms/android/build/outputs/apk/**内会看到两个发布的.apk文件：

- android-armv7-release.apk
- android-x86-release.apk

这给我们带来一个难题，我们要上传哪个到应用商店呢？
也许你已经在论坛上看到如何合并这两个apk文件了，但是幸运的是这个问题很好解决。
你可以把两个apk都提交上去，上传新版本的应用需要比之前上传的应用版本的版本号要求要高（可以在config.xml中设置）。这是提交应用后更新应用的方法，增加config.xml文件里的版本号，重新打包，重新提交。
接下来展示如何给同一个应用提交多个apk：

- 点击Upload APK按钮上传第一个apk文件，无论是android-armv7-release.apk还是android-x86-release.apk都可以
- 上传完成后点击右上角的*[Switch to Advanced Mode]*



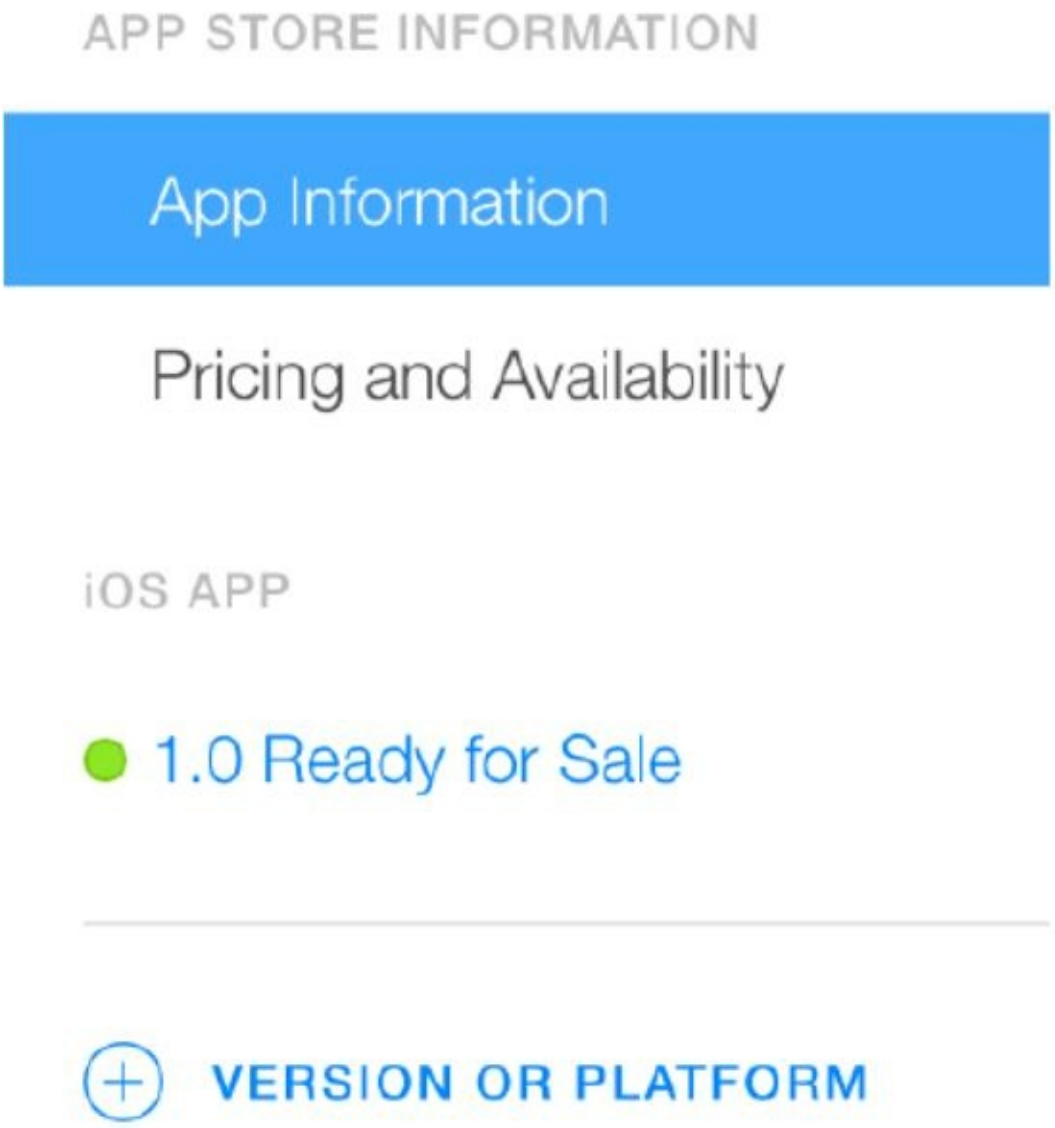
转换到高级模式后，再次点击上传按钮上传第二个apk。
这样两个apk文件都上传好了，可以在列表中看到了。可以看到他们版本号一样但是版本代号有点不用 -- 有一个的版本比另一个高 -- crosswalk自动处理。

在App商店上进行更新

只剩下这个最后的主题了。【译者：原话很长，我懒得一个字一个字的去翻译了，翻译的话比我罗嗦的这段加上前面那句还长】
当应用活在应用商店之中的时候，会需要是不是的对他更新 -- 添加一些新功能或者修改一些上线后发现的bug。
幸运的是更新应用非常简单。只需要修改config.xml文件提升版本号：

```
<widget id="com.yourname.yourproject" version="1.0.1"
    xmlns="http://www.w3.org/ns/widgets"
    xmlns:cdv="http://cordova.apache.org/ns/1.0">
```

上面的范例中我将版本号更新为1.0.1，你可以1.1.0或者2.0.0的去提升，基本都没啥问题，只要比前一个发布版版本号高就可以了。
然后重新照着之前的步骤打包应用，不过这次会更简单些，因为现在不用生成证书和其他东西了。重点是需要使用相同的东西来签名，否则是不会有用的。
当然，也不要重新创建应用商店清单。iOS的话需要登录去iTunes Connect中管理已有的应用然后提交一个新的包。你能找到更新选项的：



选择iOS选项。输入config.xml中的新版本号，更新列表中的其他需要更新的东西。【译者：原文还需要加入通过XCode或者Application Loader】上传新的包，再次将新的包加入到app store listing：

上传之后，需要重新提交审核。 对Android而言，只需要去Google Play developer console更新已有的应用信息。然后去列表的APK部分上喜欢新的apk文件（crosswalk的话两个apk都要上传）然后点击‘Publish now to Production’。
新版本需要重新走审核流程，这个流程iOS还是要一周，Google Play的话几小时。

如果你挺过来了整个课程（或者没有挺过来），我会给你一个大大的...

谢谢！

不仅是因为在这学习HTML5移动应用开发（我觉得是非常明智的选择）上投入时间和金钱，也对于你给我投入了时间和捡钱【译者：好惭愧啊，能力允许的情况下大家还是尽量买正版书】。我喜欢给开发人员教授HTML5移动开发，这对我来说意味着你买这本书是对我能力的肯定，购买这本书让我可以投入更多的时间去给开发人员制作更多内容。

如果你完成了这些课程的话，那么你应该已经很熟悉我的博客了,如果你不知道的话，可以去看看。我每周会在上面发布免费的HTML5移动开发教程，如果你想拓展你的Ionic技能的话，可以参考一下。

如果想要一些关于HTML5移动开发方面的帮助的话，请联系我。我会尽量解答所以请求，因为我收到的邮件也不少，可能无法一一解答。如果你想要一些私人咨询服务的话，请联系咨询服务。

同时，可以随时给我发邮件告诉我你在做什么，我很乐于知道HTML5开发人员都在感谢么子！