



Migration Guide



Migrating to Ionic 2 from Ionic 1

Updating your Ionic 1 app for the next generation of Ionic, Angular, and JavaScript

Migrating to Ionic 2 from Ionic 1

Introduction

Ionic 2, the next version of the popular Ionic Framework, has been in development since early 2015. The Ionic team has been working closely with the Angular team on Ionic and Angular 2, focused on the major effort to build the next generation of Angular using new JavaScript and Web technology standards. With a rare opportunity to rethink how a mobile app framework should work with the next generation of JavaScript, Ionic 2 quickly became an ambitious effort to take all the lessons learned from Ionic 1 and build the best framework for the future of mobile apps.

Today, many people still use Ionic 1 and Angular 1, by most measures two wonderful frameworks. However, there has been a massive ecosystem shift in the JavaScript world towards ES6 and TypeScript, along with embracing new technologies like Web Components. Angular 2 and Ionic 2 were built to carry web developers into the future of JavaScript.

We're incredibly proud of Ionic 2 and the many innovations and improvements that have gone into it. However, we know that upgrading and migrating any code base is a daunting task. We hope this migration guide makes the upgrade process less daunting, and encourages your team to embrace the exciting trends in mobile app development that Ionic 2 is well positioned to take advantage of.

Why Upgrade from Ionic 1 to Ionic 2?

Many developers using Ionic 1 ask: "Why should I upgrade? Ionic 1 and Angular 1 work just fine for me." Our answer to that is to please use the version that works best for you. Nothing makes us happier than to hear you are successful with any version of Ionic.

However, there are some specific reasons developers might wish to upgrade to Ionic 2:

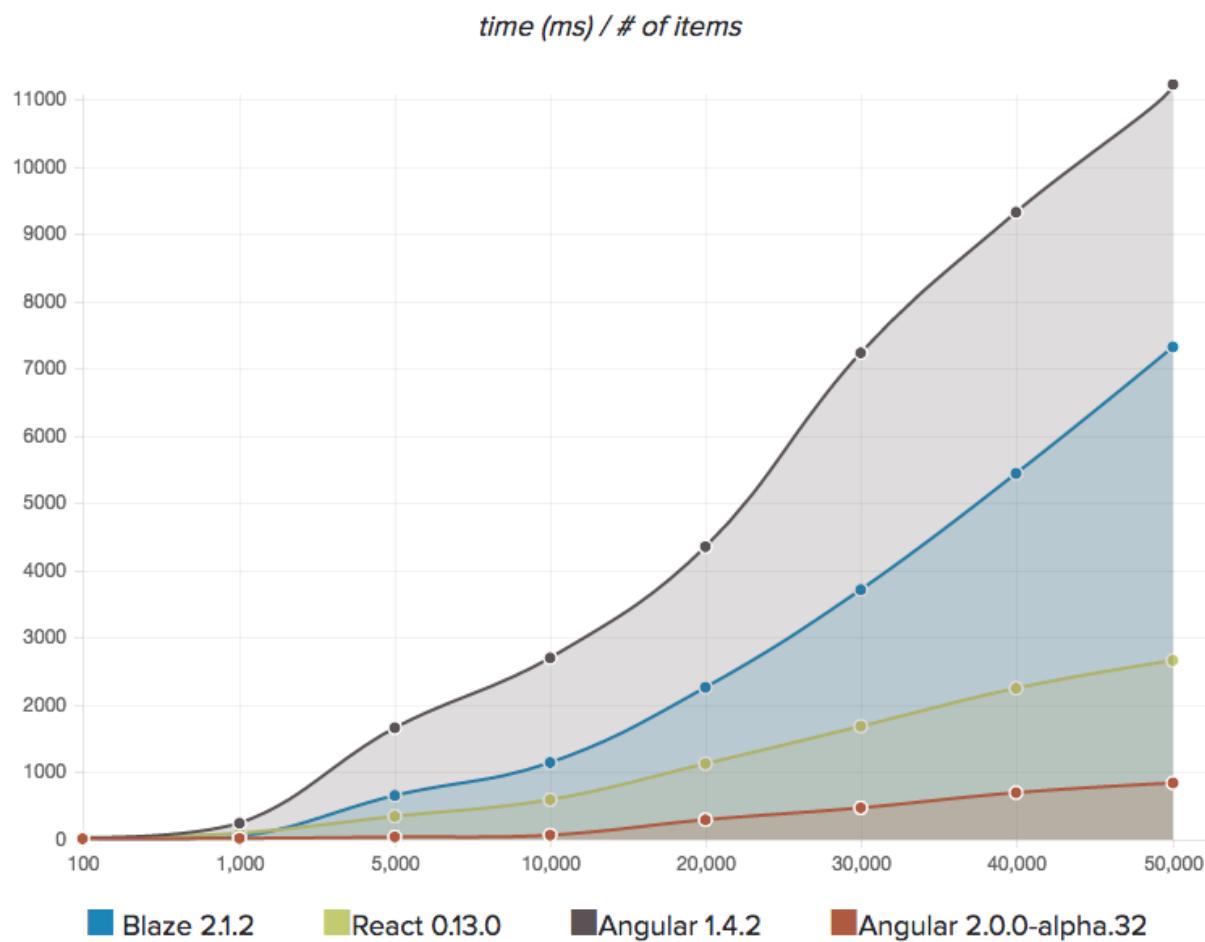
Improved Performance

Performance is an important consideration for any mobile app, Ionic or otherwise. Ionic 2 brings a number of important performance improvements you should keep in mind.

To start, Angular 2 is a faster framework than Angular 1 in a few important ways. First, change

detection uses a dramatically more efficient system where changes are isolated to components, and changes are uni-directional compared to the cyclic nature of Angular 2. This means no more infinite \$digest loops, and changes in one component that result in the entire component tree updating. It also means we avoid the treacherous two-way-binding that often ruined Angular 1 app performance.

Generate Rows: Script Times



(Angular 2 performance compared for large list rendering. [Image credit](#))

Additionally, Angular 2.0.0 brings a new pre-compilation called the [Ahead-of-Time Compiler](#) system that packs templates into a hyper-efficient format, resulting in near-instantaneous template rendering that doesn't require expensive parsing or network requests.

Beyond Angular 2 being faster, Ionic 2 has been rebuilt from the ground up for performance. Every component makes use of new APIs like `requestAnimationFrame` for 60 FPS animations, [will-change properties](#) for browser rendering optimization, and enhanced support for high-

performance native scrolling. Additionally, every component is audited for render, paint and CPU usage performance and rigorously tested.

More Components

By our count, Ionic 1 has ~15 core components. By comparison, Ionic 2 has nearly thirty high quality and well tested components, as well as countless additional APIs and features.

Some major new components include the all new Navigation component that makes it possible to navigate and link to any page in your app from anywhere, as well as flexibility in presenting pages as modal windows, or using the standard transition.

Additionally, Ionic 2 comes with powerful new Form controls that work both natively and on the web, innovating on the stock form controls we've come to know as web developers. One great example of this is the all new [Date Time picker](#).

Beyond core components, Ionic sports stronger theming capabilities and support for new native APIs like Taptic on iOS.

Native Support and Storage Utilities

Ionic 2 comes with a [powerful set of Native APIs](#) for accessing any native feature your app needs, such as Bluetooth, Health Kit, Touch ID, Fingerprint Auth, Camera, and a whole lot more.

Additionally, Ionic 2 comes with a flexible [key-value Storage system](#) that works on top of SQLite, IndexedDB, Local Storage, and other engines so you don't need to worry about the nuances of the underlying storage engine. The Storage system exposes a simple key-value API making it easy to store structured JSON data for building offline apps.

Support for Progressive Web Apps

Progressive Web Apps (PWAs) are a new paradigm shift in mobile app development, bringing all new APIs to mobile browsers to meet users where they are with improved mobile experiences and capabilities. An effort [largely started by Google](#), PWAs aim to reduce the friction for mobile experiences, especially for users coming from Google Search and web links. Instead of pushing users to download a mobile app, businesses can now provide a traditional app experience immediately to mobile web and Google Search users, with many of the features we expect from apps: installing to home screen, push notifications, offline support, and more.

With Progressive Web App support in Ionic 2, an Ionic developer can deploy a mobile app to the app stores *and* the mobile web with the exact same code. This is important because users are

coming from many different places today, and businesses can no longer afford to lose mobile web traffic through download interstitials and ads that Google says can [result in nearly 69% page abandonment rates](#). It also means Ionic developers are now even *more* cross-platform than their counterparts: one app runs natively *and* on the web with no changes! Talk about efficiency.

For further reading on PWAs, we have a number of blog posts that dive deeper into this brave new world, starting with our [What are Progressive Web Apps?](#) post.

Modern Javascript

Another great reason to migrate to Ionic 2 from Ionic 1 is to convert your ES5 Angular 1 JavaScript into modern, standards-compliant ES2015/TypeScript. Considering the JavaScript ecosystem is embracing ES2015 and TypeScript, your code will benefit from the change and your team will be able to work more effectively with the rest of the JavaScript ecosystem.

Additionally, Angular 1 required many custom operations and concepts that have standards-compliant analogs in ES2015/TypeScript. This includes \$scope which has been replaced with Class instance data, ng modules which have been replaced with ES2015 modules, services which have been replaced with vanilla classes, and many more.

In particular, enterprise teams will find TypeScript to be a boon to the maintainability and stability of their code, with its powerful type system and advanced IDE features available.

Quickly: Angular 2 vs Angular 1

This migration guide is not a proper introduction to Angular 2 or TypeScript/ES6. For that, we recommend the myriad Angular 2 tutorials available online, starting with the official [Angular 2 Quickstart](#). The Ionic team also maintains a simple introduction to some of the core concepts [in Angular 2 and ES6/TypeScript](#). Additionally, if you'd like to try TypeScript in an existing v1 project to familiarize yourself, we have an [introductory blog series](#) on the topic.

Generally, we find that developers familiar with the concepts in Angular from doing Angular 1 development are able to pick up Angular 2 quickly, as the concepts underneath are very similar. As we like to say: it's Angular all the way down.

As you convert Angular 1 code to Angular 2, here's a quick guide for mapping the old concepts to the new ones:

- Angular's custom functions for run/config/controller/directive/service have been replaced with standard ES6 and TypeScript classes or methods. \$scope is now just class

instance data. There's no more digest as Angular uses zone.js to automatically update components when instance data changes (it feels a bit like magic!)

- Angular directive usage in templates, such as ng-repeat and ng-if, now use the syntax *ngFor, *ngIf, respectfully. Other directives that dynamically set attributes, such as ng-class and ng-src, now use the dynamic attribute syntax, such as [class]="" and [src]="" . Events now use the (eventName) syntax, such as (click)="onClick(\$event)".
- Custom directives no longer use the custom bind syntax, instead using [@Input and @Output directives](#) on class data to indicate properties that can be bound to.

This is hardly an exhaustive list but should help you mentally map many of the most common features of Angular 1 to 2. The next step for learning Angular 2 concepts and how they map to Angular 1 is by reading the official [Angular 1.x Upgrade guide](#).

Step 0: Before We Jump In

This document walks through migrating an existing Ionic v1 app to v2. Follow along step-by-step, or explore the two projects side by side on github. The repos for the final v1 app and the migrated v2 app can be found here:

V1: <https://github.com/driftyco/ionic-migration-demo-v1>

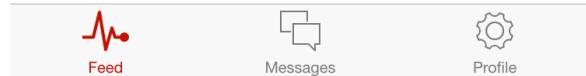
V2: <https://github.com/driftyco/ionic-migration-demo-v2>

The process for migrating an Ionic 1 app to an Ionic 2 app involves a few specific operations we'll either do once, or repeatedly. Here's an outline of what we're about to do:

1. Start a new Ionic 2 app from scratch
2. Convert our main app structure (i.e. tabs/menu/etc.)
3. Update our theme
4. Convert each Ionic 1 view/controller to an Ionic 2 Page, including porting our templates
5. Convert each Ionic 1 service/factory to an Ionic 2 Provider
6. Convert custom Angular 1 code to standards-compliant ES6/TypeScript
7. Update routes to use new navigation and deeplinks

There may be a few tasks you need to do that are unique to your app, but generally speaking this is the process we will take and, chances are, you'll be pleasantly surprised how straightforward moving from Ionic 1 to Ionic 2 is.

For this migration guide, we are going to use an Ionic app that renders a simple social network, called Ion Friends:



Throughout the document, we will be showing code snippets for both our v1 apps and our v2 app, and will denote each with these labels. The Ionic 1 label refers to our existing v1 app, and the Ionic 2 label refers to the new one we are porting over to.

Ionic 1

Ionic 2

Additionally, we will use monospace font to refer to commands that should be typed into a command prompt.

Finally, please keep in mind that we won't be porting all the features over for the sake of this example, but will teach the migration concepts you can use to apply to your app.

Step 1: Create a new Ionic 2 App

While it's possible to update an Ionic 1 app in-place, we recommend starting a fresh app to make sure you have the proper files required for building apps and to avoid perplexing build issues with two Ionic versions. Once your app works in the new Ionic 2 shell, it can be moved back to your existing app repo to replace your existing code.

To speed things up, if your app uses tabs or menu, you can create a new starter app that has that layout by default. If your app is custom, you can start with a blank layout and build it from scratch.

To start, let's install Ionic (or make sure you are using the latest version of the Ionic CLI), create a new app, and start an ionic serve session. For this example, we will create a new tabs-based layout. Menu based apps would be "sidemenu."

```
npm install -g ionic  
ionic start app tabs --v2  
cd app  
ionic serve
```

This will start a new app with the "tabs" template. To keep things simple, content has been removed from the default templates in the screenshots proceeding.

Now that we have our v2 app, let's go back to the v1 app and start the process of porting over our code.

Step 2: Convert Main App Structure

Chances are, your app has a high level layout style that we can adapt over to Ionic 2. For example, you might have a Tabbed layout, or a Side Menu, or both together!

In Ion Friends, our app has a tabs layout as the main layout of the app.



If you followed the start example above, we now have a tabs template available in `src/pages/tabs/tabs.html`. If not, create a new tabs page with `ionic g page tabs` and then add the code below.

Let's compare our v2 and v1 versions to see the similarities in structure:

src/pages/tabs/tabs.html

```
<ion-tabs>
  <ion-tab [root]="tab1Root" tabTitle="Home" tabIcon="home"></ion-tab>
  <ion-tab [root]="tab2Root" tabTitle="About"
    tabIcon="information-circle"></ion-tab>
  <ion-tab [root]="tab3Root" tabTitle="Contact" tabIcon="contacts"></ion-tab>
</ion-tabs>
```



Back in our v1 code, open www/templates/tabs.html.

```
<ion-tabs class="tabs-icon-top tabs-color-active-positive">

  <!-- Dashboard Tab -->
  <ion-tab title="Feed" icon-off="ion-ios-pulse" icon-on="ion-ios-pulse-strong" href="#/tab/dash">
    <ion-nav-view name="tab-feed"></ion-nav-view>
  </ion-tab>

  <!-- Chats Tab -->
  <ion-tab title="Messages" icon-off="ion-ios-chatboxes-outline" icon-on="ion-ios-chatboxes" href="#/tab/chats">
    <ion-nav-view name="tab-chats"></ion-nav-view>
  </ion-tab>

  <!-- Account Tab -->
  <ion-tab title="Profile" icon-off="ion-ios-gear-outline" icon-on="ion-ios-gear" href="#/tab/account">
    <ion-nav-view name="tab-account"></ion-nav-view>
  </ion-tab>

</ion-tabs>
```



The v2 tabs and v1 tabs are incredibly similar. In fact, the structure is the exact same, though some of the attributes have changed and we need less boilerplate to set our tabs up.

Updating our v2 tabs to match the titles and icons of our v1 tabs, gives us this:

```
<ion-tabs>
  <ion-tab [root]="tab1Root" tabTitle="Feed" tabIcon="pulse"></ion-tab>
  <ion-tab [root]="tab2Root" tabTitle="Messages" tabIcon="chatboxes"></ion-tab>
  <ion-tab [root]="tab3Root" tabTitle="Profile" tabIcon="cog"></ion-tab>
</ion-tabs>
```

Updating our Pages

Our new v2 app comes with three default tab pages (“home”, “contact”, “about”) in v2/src/pages. We can either rename those and each template file and class reference to match the ones from our v1 app, or remove them add generate 4 new pages. Let’s do the latter as it’s faster:

```
rm -rf src/pages/home src/pages/about src/pages/contact
ionic g page feed
ionic g page messages
ionic g page message-detail
ionic g page profile
```

We now have four new pages in src/pages with the proper titles and classes for our feed, messages, and profile tabs, along with a message detail page. The last step is to update our app module so Angular knows about the new pages.

Edit src/app/app.module.ts and update the references to the new pages:

```
import { NgModule, ErrorHandler } from '@angular/core';
import { IonicApp, IonicModule, IonicErrorHandler } from 'ionic-angular';
import { MyApp } from './app.component';
import { ProfilePage } from '../pages/profile/profile';
import { MessagesPage } from '../pages/messages/messages';
import { FeedPage } from '../pages/feed/feed';
import { TabsPage } from '../pages/tabs/tabs';
import { MessageDetailPage } from '../pages/message-detail/message-detail';

@NgModule({
  declarations: [
    MyApp,
    ProfilePage,
    MessagesPage,
    FeedPage,
    TabsPage,
    MessageDetailPage
  ],
  imports: [
    IonicModule.forRoot(MyApp)
  ],
  bootstrap: [IonicApp],
  entryComponents: [
    MyApp,
    ProfilePage,
    MessagesPage,
    FeedPage,
    TabsPage,
    MessageDetailPage
  ],
  providers: [
    {provide: ErrorHandler, useClass: IonicErrorHandler}
  ]
})
export class AppModule {}
```

Lastly, the tabs component needs to be updated to reference the new pages we created:

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

import { ProfilePage } from '../profile/profile';
import { MessagesPage } from '../messages/messages';
import { FeedPage } from '../feed/feed';

@Component({
  templateUrl: 'tabs.html'
})
export class TabsPage {
  // this tells the tabs component which Pages
  // should be each tab's root Page
  tab1Root: any = FeedPage;
  tab2Root: any = MessagesPage;
  tab3Root: any = ProfilePage;

  constructor() {

  }
}
```

Now our app should look like this:



Step 3: Update Theme

Updating the theme after the app structure is in place helps our app start to look recognizable, and it's fun to build with our branding.

Let's bring over our red theme to our new Ionic 2 app:



In our v1 app, our theme SCSS is in scss/ionic.app.scss.

We've changed the color of the \$positive variable to be our red color:

```
$positive: #c41200;
```

ionic 2

In our Ionic 2 app, open `src/theme/variables.scss`, and find the `$colors` variable. We're going to modify the value for "primary" which, in Ionic 2, sets our color for many common elements (buttons, highlights, etc.).

Additionally, we'll set two more variables to change the default header color and update the tabs color so we have a light background with our red highlight for the icons:

```
$colors: (
  primary:    #c41200,
  secondary:  #32db64,
  danger:     #f53d3d,
  light:      #f4f4f4,
  dark:       #222
);

$toolbar-background: #c41200;
$tabs-ios-background: #f4f4f4;
$tabs-md-background: #f4f4f4;
```

Step 4: Convert Ionic 1 View/Controller to an Ionic 2 Page

Next, we will convert our Ionic 1 Views/Controllers into Ionic 2 Pages by updating the page component classes and their corresponding view templates.

ionic 1

In our example, our controllers are in `'www/js/controllers.js'`, and our view templates are in `www/templates/`. Let's start with the controllers.

The first controller we see is FeedCtrl, which is the main controller for the Feed page:

```
angular.module('starter.controllers', [])

.controller('FeedCtrl', function($scope) {
  $scope.items = [
    {
      name: 'Marty McFly',
      date: 'November 05, 1955',
      avatar: 'img/mcfly.jpg',
      image: 'img/delorean.jpg',
      content: 'Wait a minute. Wait a minute, Doc. Uhhh...
DeLorean?! Whoa. This is heavy.'
    }
  ];
})
```

And the corresponding template:

```
<ion-view view-title="Feed">
  <ion-content class="padding">
    <ion-list>
      <div class="list card" ng-repeat="item in items">
        <ion-item class="item item-avatar">
          
          <h2>{{item.name}}</h2>
          <p>{{item.date}}</p>
        </ion-item>

        <ion-item class="item item-body">
          
          <p>
            {{item.content}}
          </p>
          <p>
            <a href="#" class="subdued">1 Like</a>
            <a href="#" class="subdued">5 Comments</a>
          </p>
        </ion-item>
      </div>
    </ion-list>
  </ion-content>
</ion-view>
```



Open `src/pages/feed/feed.ts`, and we are going to take the scope data for our items list and change that to standard class instance data:

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

@Component({
  selector: 'page-feed',
  templateUrl: 'feed.html'
})
export class FeedPage {
  items: any[];

  constructor(public navCtrl: NavController) {
    this.items = [
      {
        name: 'Marty McFly',
        date: 'November 05, 1955',
        avatar: 'img/mcfly.jpg',
        image: 'img/delorean.jpg',
        content: 'Wait a minute. Wait a minute, Doc. Uhhh... DeLorean?! Whoa. This is heavy.'
      }
    ];
  }
}
```

A few things to note here. Instead of Angular 1's custom \$scope concept, we use instance data on the class for the page. We also defined our items as an array of "any" type items so typescript understands this is a defined object on FeedPage.

The structure of our code looks a bit different from v1. One way to think about the change is Angular 1's controller and directive combine into one "Component" in Angular 2 that contain both the logic for the page and the template. Additionally, where we injected \$scope into our controller in v1, in v2 we inject needed Ionic/Angular 2 services into the constructor.

Finally, here's the updated v2 template in src/pages/feed/feed.html:

```
<ion-header>

  <ion-navbar>
    <ion-title>Feed</ion-title>
  </ion-navbar>

</ion-header>

<ion-content>
  <ion-list>
    <ion-card *ngFor="let item of items">
      <ion-item>
        <ion-avatar item-left>
          <img [src]="item.avatar">
        </ion-avatar>
        <h2>{{item.name}}</h2>
        <p>{{item.date}}</p>
      </ion-item>

      <img [src]="item.image">

      <ion-card-content>
        <p>
          {{item.content}}
        </p>
      </ion-card-content>

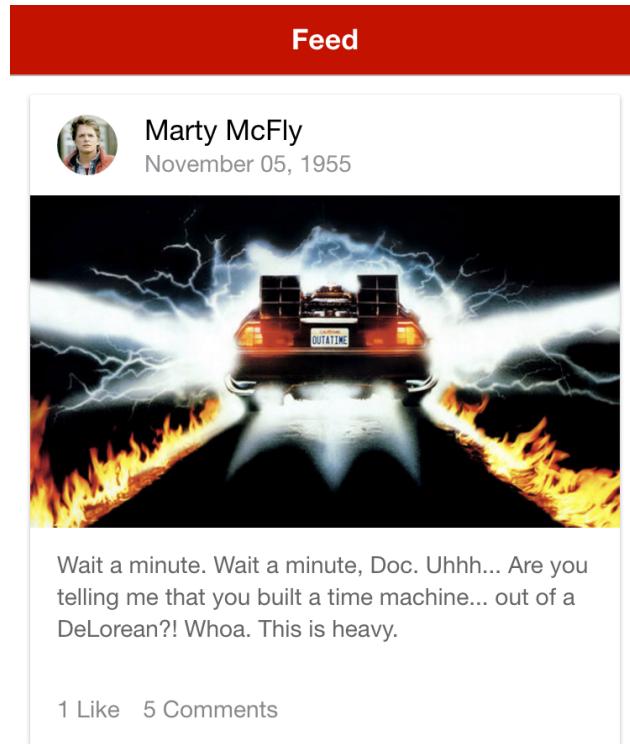
      <ion-item>
        <p>
          <a href="#" class="subdued">1 Like</a>
          <a href="#" class="subdued">5 Comments</a>
        </p>
      </ion-item>
    </ion-card>
  </ion-list>
</ion-content>
```

Compare the v1 and v2 templates side by side. Notice a few similarities: both use `<ion-content>`, `<ion-list>`, and `<ion-item>` just like before. In Ionic 2, we've largely carried over the functionality of these core components, and they are used just like they were in v1.

One big difference is that we have an `<ion-header>` and `<ion-navbar>` directly in our template. This makes it easy to customize the navbar for our page, compared to v1 where those settings had to be done outside of the page. Also, we no longer use `<ion-view>` as a wrapping component because it's not necessary, the page component class is now the "view."

For other components like cards, buttons, etc. we can compare our components with the v2 component list and adjust to the new syntax. Thankfully, you'll find that v2 has all the same components, even though they might have slightly different syntax.

Finally, we need to move our assets from v1/www/img to v2/src/assets/img. After we do this, our app will look like this:



We'll follow the same process for converting the messages and profile tabs.

Step 5: Convert Providers/Services



Our Ion Friends app has a factory in `www/js/services.js`:

```
angular.module('starter.services', [])

.factory('Chats', function() {
  // Might use a resource here that returns a JSON array

  // Some fake testing data
  var chats = [
    {
      id: 0,
      name: 'Ben Sparrow',
      lastText: 'You on your way?',
      face: 'img/ben.png'
    }, {
      id: 1,
      name: 'Max Lynx',
      lastText: 'Hey, it\'s me',
      face: 'img/max.png'
    }, {
      id: 2,
      name: 'Adam Bradleyson',
      lastText: 'I should buy a boat',
      face: 'img/adam.jpg'
    }, {
      id: 3,
      name: 'Perry Governor',
      lastText: 'Look at my mukluks!',
      face: 'img/perry.png'
    }, {
      id: 4,
      name: 'Mike Harrington',
      lastText: 'This is wicked good ice cream.',
      face: 'img/mike.png'
    }];
}

return {
  all: function() {
```



To map this factory to our Ionic 2 project, let's create a new provider:

```
ionic g provider chats
```

Which creates a new file in src/providers/chats.ts:

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

@Injectable()
export class Chats {
  constructor() {

  }
}
```

As we see, instead of having a custom factory we have a plain ES6/TypeScript class, with the `@Injectable` decorator that tells Angular this is a class that can be used with dependency injection.

Next, we need to add this provider to our Angular NgModule in src/app/app.module.ts so that Angular knows how to inject it into our pages and components:

```
import { Chats } from '../providers/chats';

@NgModule({
  declarations: [
    MyApp,
    ProfilePage,
    MessagesPage,
    FeedPage,
    TabsPage,
    MessageDetailPage
  ],
  imports: [
    IonicModule.forRoot(MyApp)
  ],
  bootstrap: [IonicApp],
  entryComponents: [
    MyApp,
    ProfilePage,
    MessagesPage,
    FeedPage,
    TabsPage,
    MessageDetailPage
  ],
  providers: [
    Chats,
    {
      provide: ErrorHandler, useClass: IonicErrorHandler
    }
  ]
})
export class AppModule {}
```

Going back to the Chats provider, after adding our chat data to this class, we now have:

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

@Injectable()
export class Chats {
  chats: any[];

  constructor() {
    this.chats = [{  
      id: 0,  
      name: 'Ben Sparrow',  
      lastText: 'You on your way?',  
      face: 'img/ben.png'  
}, {  
      id: 1,  
      name: 'Max Lynx',  
      lastText: 'Hey, it\'s me',  
      face: 'img/max.png'  
}, {  
      id: 2,  
      name: 'Adam Bradleyson',  
      lastText: 'I should buy a boat',  
      face: 'img/adam.jpg'  
}, {  
      id: 3,  
      name: 'Perry Governor',  
      lastText: 'Look at my mukluks!',  
      face: 'img/perry.png'  
}, {  
      id: 4,  
      name: 'Mike Harrington',  
      lastText: 'This is wicked good ice cream.',  
      face: 'img/mike.png'
```

Finally, updating our Messages page in `src/pages/messages/messages.ts` to include the Chats provider and an event handler to open the chat when the user taps on it:

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

import { MessageDetailPage } from '../message-detail/message-detail';

import { Chats } from '../../providers/chats';

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

  constructor(public navCtrl: NavController, public chats: Chats) {}

  openItem(chat) {
    this.navCtrl.push(MessageDetailPage, {
      chat: chat
    })
  }
}
```

Now our Chats provider is available as `this.chats`, as TypeScript's `public` keyword in the constructor is shorthand for `this.chats = chats`.

`openItem(chat)` is called when the user taps on a message in the message list. We then `push` an instance of the `MessageDetailPage` with our chat as a param onto our `NavController` stack, causing the app to transition to the new page.

Here's what our updated `messages.html` template looks like with the new click handler and migrated template markup:

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

<ion-content>
  <ion-list>
    <ion-item-sliding *ngFor="let chat of this.chats.all()">
      <button ion-item (click)="openItem(chat)">
        <ion-avatar item-left>
          <img [src]="chat.face">
        </ion-avatar>
        <h2>{{chat.name}}</h2>
        <p>{{chat.lastText}}</p>
      </button>
      <ion-item-options>
        <button ion-button color="danger" (click)="removeItem(chat)">
          Delete
        </button>
      </ion-item-options>
    </ion-item-sliding>
  </ion-list>
</ion-content>
```

Step 6: Convert Ionic Run/Config/Root Controller

By default, Ionic 1 apps come with some high-level app initialization code in `www/js/app.js`, often in an Angular run or config function. This is stuff that we need to run for the entire app.

In Ionic 2, there is no concept of root run code. Instead the root of our app is a component that often will contain app initialization and the initial view logic. In Ionic 2, our entire app is made of components!



Open `www/app.js`.

The first thing we see in this file are our main app module imports.

These will be adjusted into ES2015/TypeScript imports as shown in previous examples, and the pattern of using strings like 'starter.controllers' doesn't have an analog in Ionic 2/Angular 2/TypeScript.

```
angular.module('starter', ['ionic', 'starter.controllers', 'starter.services'])

.run(function($ionicPlatform) {
  $ionicPlatform.ready(function() {
    // Hide the accessory bar by default (remove this to show the accessory bar above the keyboard
    // for form inputs)
    if (window.cordova && window.cordova.plugins && window.cordova.plugins.Keyboard) {
      cordova.plugins.Keyboard.hideKeyboardAccessoryBar(true);
      cordova.plugins.Keyboard.disableScroll(true);

    }
    if (window.StatusBar) {
      // org.apache.cordova.statusbar required
      StatusBar.styleDefault();
    }
  });
})

.config(function($stateProvider, $urlRouterProvider) {
```

In the run method, our app is performing some initial setup, such as configuring our status bar and the keyboard settings.



Converting that to Ionic 2, we simply update the constructor of our app component in src/app/app.component.ts:

```
import { Component } from '@angular/core';
import { Platform } from 'ionic-angular';
import { StatusBar, Splashscreen } from 'ionic-native';

import { TabsPage } from '../pages/tabs/tabs';

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

  constructor(platform: Platform) {
    platform.ready().then(() => {
      // Okay, so the platform is ready and our plugins are available.
      // Here you can do any higher level native things you might need.
      StatusBar.styleDefault();
      Splashscreen.hide();

      // Keyboard stuff uses new defaults, no need to have the same config code
    });
  }
}
```

There's no need for a run function anymore, just handle it in the constructor of the entry component, which is our MyApp class!

Step 7: Update Routing

In Ionic 1, routing used a custom version of ui-router to map URLs to Ionic views. In Ionic 2, we've taken a more native-style approach to navigation using push/pop style navigation without being bound to URLs.



For example, this was the routing config we used in Ionic 1:

```
.config(function($stateProvider, $urlRouterProvider) {

  // Ionic uses AngularUI Router which uses the concept of states
  // Learn more here: https://github.com/angular-ui/ui-router
  // Set up the various states which the app can be in.
  // Each state's controller can be found in controllers.js
  $stateProvider

    // setup an abstract state for the tabs directive
    .state('tab', {
      url: '/tab',
      abstract: true,
      templateUrl: 'templates/tabs.html'
    })

    // Each tab has its own nav history stack:
    .state('tab.feed', {
      url: '/feed',
      views: {
        'tab-feed': {
          templateUrl: 'templates/tab-feed.html',
          controller: 'FeedCtrl'
        }
      }
    })

    .state('tab.chats', {
      url: '/chats',
      views: {
        'tab-chats': {
          templateUrl: 'templates/tab-chats.html',
          controller: 'ChatsCtrl'
        }
      }
    })
    .state('tab.chat-detail', {
      url: '/chats/:chatId',
      views: {
        'tab-chats': {
          templateUrl: 'templates/chat-detail.html',
          controller: 'ChatDetailCtrl'
        }
      }
    })
})
```

In Ionic 2, pages can be navigated to from anywhere and are not strictly bound to a URL. This is how traditional native SDKs work.

To get to a page in Ionic 2, we either set it as the root of or push it onto a NavController. A NavController is at the core of <ion-nav>, and individual <ion-tab>'s. This means each tab has its own parallel navigation, making it easy to switch between each tab while preserving our place in the stack.

For example, in our demo above, we set the [root] of each tab to the page corresponding to our Feed, Messages, and Profile pages, respectfully. We could easily change the root to a different page.

```
<ion-tabs>
  <ion-tab [root]="tab1Root" tabTitle="Feed" tabIcon="pulse"></ion-tab>
  <ion-tab [root]="tab2Root" tabTitle="Messages" tabIcon="chatboxes"></ion-tab>
  <ion-tab [root]="tab3Root" tabTitle="Profile" tabIcon="cog"></ion-tab>
</ion-tabs>
```

Then, inside of MessagesPage, we grab the reference to our parent NavController and, on a click event of one of our messages, push an instance of MessageDetailPage onto it to navigate to the detail page for the message:

```
export class MessagesPage {

  constructor(public navCtrl: NavController, public chats: Chats) {}

  openItem(chat) {
    this.navCtrl.push(MessageDetailPage, {
      chat: chat
    })
  }
}
```

Any page inside of a NavController has access to their parent NavController through dependency injection, and a page does not need to know where it lives in the app, making it easy to navigate to a page anywhere inside of any NavController.

For a full introduction to NavController and Ionic Navigation, see the [NavController docs](#).

Deeplinking

Though Ionic navigation no longer relies on strict routing, we can still identify each page with a unique URL for deeplinking purposes. This makes it easy to open our app to a specific page through a deeplink.

To do this, we add a `links` option to our Ionic module import in `app.module.ts`:

```
@NgModule({
  declarations: [
    MyApp,
    ProfilePage,
    MessagesPage,
    FeedPage,
    TabsPage,
    MessageDetailPage
  ],
  imports: [
    IonicModule.forRoot(MyApp, {}, {
      links: [
        { component: MessagesPage, name: 'Messages', segment: 'messages' },
        { component: MessageDetailPage, name: 'MessageDetail', segment: 'message/:chatId' }
      ]
    })
  ],
})
```

Then, for example, when our `MessageDetailPage` is navigated to, the URL bar will have `message/:chatId` where `chatId` is the id of the chat object sent as nav params.

Extra: Convert ngCordova to Ionic Native

If you're using ngCordova for native plugins, we've built a new, framework-agnostic native plugin library called [Ionic Native](#) that has even more plugins, proper types for TypeScript, support for promises and observables, and new documentation.

To swap in Ionic Native for ngCordova, find your plugin in our [Ionic Native docs](#), and change your references from `$cordovaPLUGIN` to the class reference. In general, the method names and response types are the same, so changing over should be a small task.

Here's an example of moving from `$cordovaCamera` in ngCordova to `Camera` in Ionic Native:

⌚ ionic 1

```
$cordovaCamera.getPicture({}).then(function(imageURI) {  
}, function(err) {  
});
```

⌚ ionic 2

```
import { Camera } from 'ionic-native';  
  
Camera.getPicture({}).then((imageURL) => {  
}, (err) => {  
})
```

Extra: Convert build scripts to v2

Ionic 2 no longer ships with a user-configurable gulpfile, in lieu of an npm-installed package of pre-fab build scripts for Ionic we call [Ionic App Scripts](#).

The benefit of this change is that the Ionic team can now develop advanced build tool features, like our recently launched build and runtime error reporting.

Conclusion

Ionic 2 carries over many of the same concepts from Ionic 1, including many components with identical or very similar usage and features, and relies on many of the same tools used for Ionic 1 development, such as the Ionic CLI. Plus, Ionic 2 is built by the exact same team that built Ionic 1.

Additionally, developers will find that Angular 2 is based on the same concepts they know and love from Angular 1, such as familiar template directives and Angular's data-driven approach. Once Angular 1 developers start using Angular 2, they will find they already understand the core concepts of Angular 2.

Moving from Ionic 1 to Ionic 2 is more than just moving to a new version of Ionic, it also brings an opportunity to evolve code written for the last generation of JavaScript into the new, modern JavaScript standard, while benefitting from advanced build tools and the powerful type system

in TypeScript. Additionally, it's an opportunity for software teams and businesses to be best positioned for the exciting new world of Progressive Web Apps and converting customers faster than was possible in the app store era alone.

The company behind Ionic 2 is growing quickly and is well funded by some of the top investors in the world. Ionic 2 is our vision for a future of mobile app development that is cheaper, faster, and more accessible for developers around the world.

And we're just getting started!