***Giáo trình***

**Lập trình Ứng dụng Di động**

**GS. Phan Viết Hoàng**

**Bài học 4 (Buổi 4)**

**Lập trình di động với Ionic**

# References

# ‘Building Mobile App with Ionic 2’*, Josh Morony, 2016*

# Nền tảng Ionic

**Phân tích dự án Ionic**

**(Lesson 2: Anatomy of An Ionic 2 Project)**

Now that we’ve covered how to get Ionic 2 installed and how to generate a project, I want to cover what the various files and folders contained within your newly generated project do. When you create a blank Ionic 2 application, your folder structure will look like this:

At first glance, there’s an intimidating amount of stuﬀ there - but there’s really not that much you need to

worry about, and it will all make total sense with a little explanation. We’re going to discuss what everything does, but I will cover the important stuﬀ first in more detail (mainly the files and folders that you will be modifying) and then cover the less important stuﬀ.

**Important Files & Folders**

These files and folders are ones that you will be using on a frequent basis, so it’s important that you understand their role. Fortunately, there’s not too many of them!

**src**

This is where most of the action occurs. In a default application, your **src** folder will contain:

* An **app** folder
* A **pages** folder
* A **theme** folder
* An **assets** folder
* An **index.html** file

The **pages** folder will contain all of the page components for your application. If you look inside of the **pages** folder you should see another folder called **home**. This is a **component**, and is made up of a classdefinition (home.ts), a template (home.html), and style definitions (home.scss). For every page in your application you will add another folder here (we can actually get the Ionic CLI to do this for us automatically), so as well as the home folder you may also have **login**, **intro**, **checkout**, **about** and many more.

The **theme** folder will contain all of the **.scss** files which define application wide styles for the application. This includes a shared variables file that contains variables you can override, and a global file that can store generic styles for the whole application. There is a whole section later in this book dedicated to theming Ionic 2 applications so we’ll discuss this in more detail there.

The **app** folder contains the **root component** for your application, **app.component.ts**. Again, we discuss

what the root component is and what it does in detail in other parts of this book so I won’t cover much right now, but essentially it’s the “starting point” for your application. You will also find a **app.module.ts** which makes use of Angular 2’s @NgModule which allows us to set up all of our application dependencies, like components and services we are using. Finally, you also have a main.ts file, this is what kicks oﬀ the bootstrapping process for your application.

Bootstrapping basically means setting up the **root component** of your application, which is the first com-ponent that is created that will then go on and create all of the rest. This creates a tree-like structure of components, with the root component at the beginning.

The **assets** folder is where you can store any static assets your application may want to make use of, like images and JSON files. These will be copied over to the build folder (so it’s important you place assets here, **not** in the www/assets folder) when your application is built. You could for ex-ample create an images folder in here, and then reference those images in your application through assets/images/myImage.png.

Although you won’t often have to, you can also edit the **index.html** file for your application in the **src** folder, and like the **assets** this will also be copied over to the build directory.

As well as the default folders that your app will contain, as you start building your application you will likely see a few more folders in here as well, but we’ll get to that later.

**IMPORTANT:** This is a really important concept to understand so make sure you read this over a coupleof times if you don’t get it initially. In the introduction section of this book we talked about webpack, transpiling and a bunch of other fancy ES6 stuﬀ - the important thing to remember is that the ES6 and TypeScript features we are using aren’t actually supported by browsers yet, so we need to “transpile” them into valid ES5 code. When you run or build your application, Ionic will bundle up everything in this app folder, perform all the magic it needs to perform, and then spits it out into the **www** folder.

When you are viewing your application through the browser you are actually viewing the bundled version inside of the **www** folder, not the code you created in the **app** folder. Likewise, when you build your application for release on iOS and Android (which we will discuss later) it is this **www** folder that will be used, not the **app** folder. **DO NOT EDIT CODE IN THE WWW FOLDER** - any changes you make in there (which is a bad idea anyway) will be overwritten when your app is rebuilt.

Using Angular 2 and the new ES6 syntax means the project structure and build process is quite a bit more complicated, but fortunately for us Ionic handles basically all of it for us. So don’t worry too much about it, this is just something important to keep in mind so that you know what is going on.

**config.xml**

You won’t need to edit this file very often, but it is a very important file. The **config.xml** file is basically used as a way to tell Cordova how to build your application for iOS and Android. You’ll have to supply some important configuration information in here which we will discuss later, but you can also define some preferences in here (like whether the splash screen should auto hide, whether the app should be portrait only, and a bunch more). You will usually only really worry about this file when you’re starting to look at getting your app on real devices.

**The Less Important Stuﬀ**

Obviously everything in your Ionic project plays a role, otherwise it wouldn’t be there. But some of it is for more advanced use cases you might not need to worry about, and some of it is just pure configuration stuﬀ that you’ll never have to touch.

**Configuration Files**

If you take a look in the root folder of your generated project you will see a bunch of configuration files.

Aside from the **config.xml** file which we discussed above, you can pretty safely just ignore all of these.

The only file you may want to edit is the **package.json** file to update the version of Ionic you are using.

**resources**

This folder simply contains the splash screens and icons that will be used for iOS and Android when you build your application. I’ll show you a really easy way to create these resources with the Ionic CLI later.

**hooks**

Hooks are used as part of your applications build process, and you can add custom scripts in here to hook into various parts of the build process. At a beginner level you likely will not need to touch this at all, but if you wanted to start developing more complicated (but useful) workflows, related to versioning or deployment of your application for example, you could create some “hooks” here.

**node\_modules**

This is another folder that you won’t have to touch at all, but it’s where all the goodies are stored. If you take a look in this folder you will find things like **ionic-angular**, **angular2**, and **ionicons**. This is where all of the source files for the various libraries that your application depends on are stored (including the Ionic framework itself).

**Lệnh Ionic**

**(Lesson 3: Ionic CLI Commands)**

The Ionic CLI is a super powerful tool - we’ve already gone through how to use it to generate a new project and display your application in the browser, but there’s a bunch more commands you should know about too, so let’s go through some of them. This is by no means an exhaustive list, but it will cover all of the commands you should be using frequently.

I’m going to list out a few commands now and what they do, and since some commands have multiple diﬀerent arguments that you can supply to them I will be using the following syntax:

ionic command [option1|option2]

in place of:

ionic command option1

This will allow you to add Cordova plugins to your project, simply supply the plugin name.

ionic build [ios|android]

When you are ready to build your application for iOS and Android you can simply run these commands to

build your project. There’s a little bit more to it than this to actually get it on your device and then submit it to App Stores, but we will discuss all that later.

ionic run [ios|android]

If you want to test your application on a real device you can use this command to deploy the app directly to your device. You can also use the --device flag to force it to run on a device that is plugged in via USB. For more information on this, please see the ‘Testing & Debugging’ section later in the book.

ionic emulate [ios|android]

Instead of deploying your application to a real device, this will launch an emulator on your machine and run it there instead.

ionic g [page|component|directive|pipe|provider|tabs] NameGoesHere

This is the generate command and it is one of my favourites, it’s a real time saver. As I mentioned before, your Ionic project will contain a bunch of components like the **home** page. To create more components you can manually create a new folder and add all the required files, or you can just run the ionic generate command to do it automatically for you, with some handy boilerplate code in place. As well as pages, this command can also generate generic components, directives, pipes, providers and tabs. I’d recommend using it for any new component you are adding to your application.

ionic plugin list

This will allow you to see a list of all the plugins you have installed in your project.

ionic plugin rm [plugin]

This will allow you to remove any plugin from your project by supplying the plugin name.

ionic platform rm [android|ios]

This will allow you to remove a platform that you have previously added.

As I mentioned, this is not an exhaustive list but it does cover the most commonly used commands. I

rarely use any commands outside of these.

# Lập trình Ionic

**Class hổ trợ**

**(Lesson 4: Decorators)**

Each class (which we will talk about in the next section) you see in an Ionic 2 application will have a **decorator**. A decorator looks like this:

**@Component**({

someThing: 'somevalue',

someOtherThing: [Some, Other, Values]

})

They definitely look a little weird, but they play an important role. Their role in an Ionic 2 application is to provide some *metadata* about the class you are defining, and they always sit directly above your class definition (again, we’ll get to that shortly) like this:

@Decorator({

*/\*meta data goes here\*/*

})

**export class** MyClass {

*/\*class stuff goes here\*/*

}

This is the only place you will see a decorator, they are used purely to add some extra information to a class (i.e. they “decorate” the class). So let’s talk about exactly how and why we would want to use these decorators in an Ionic 2 application.

The decorator name itself is quite useful, here’s a few you might see in an Ionic 2 application:

* @Component
* @Pipe
* @Directive

We can supply an object to the decorator to provide even more information on what we want. Here’s the most common example you’ll see in your applications:

**@Component**({

selector: 'page-home',

templateUrl: 'home.html'

})

**export class** HomePage {

}

Now this class knows where it needs to fetch its template from, which will determine what the user will actually see on the screen (we’ll be getting into that later as well). If you’ve got a super simple template, maybe you don’t even want to have an external template file, and instead define your template like this:

**@Component**({

selector: 'page-home',

template: `<p>Howdy!</p>`

})

**export class** HowdyPage {

}

Some people even like to define large templates using template. Since ES6 supports using backticks (the things surrounding the template above) to define multi-line strings, it makes defining large templates like this a viable option if you prefer (rather than doing something ugly like concatenating a bunch of strings).

Now that we’ve covered the basics of what a decorator is and what it does, let’s take a look at some specifics.

**Common Decorators in Ionic 2 Applications**

There are quite a few diﬀerent decorators that we can use. In the end, their main purpose is simply to describe *what* the class we are creating *is*, so that it knows what needs to be imported to make it work.

Let’s discuss the main decorators you are likely to use, and what the role of each one is. We’re just going to be focusing on the decorator for now, we will get into how to actually build something useable by defining the class in the next section.

**@Component**

I think the terminology of a *component* can be a little confusing in Ionic 2. As I mentioned, our application is made up of a bunch of components that are all tied together. These components are contained within folders inside of our main **src** folder, and they look like this:

**home**

* home.ts
* home.html
* home.scss

A **@Component** is not specific to Ionic 2, it is used generally in Angular 2. A lot of the functionality provided by Ionic 2 is done through using components. In Ionic 2, for example, you might want to create a search bar, which you could do by using one of the components that Ionic 2 provides like this:

<ion-searchbar></ion-searchbar>

You simply add this custom tag to your template. Ionic 2 provides a lot of components but you can also create your own custom components, and the decorator for that might look something like this:

**@Component**({

selector: 'my-cool-component'

})

which would then allow you to use it in your templates like this:

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

**NOTE:** Technically speaking a component should have a class definition and a template. Things like pipesand providers aren’t viewed on the screen so have no associated template, they just provide some additional functionality. Even though these are not technically components you may often see them referred to as such, or they may also be referred to as services or providers.

**@Directive**

The **@Directive** decorator allows you to create your own custom directives. Typically, the decorator would look something like this:

**@Directive**({

selector: '[my-selector]'

})

Then in your template you could use that selector to trigger the behaviour of the directive you have created by adding it to an element:

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

It might be a little confusing as to when to use **@Component** and **@Directive**, as they are both quite similar. The easiest thing to remember is that if you want to modify the behaviour of an existing component use a **directive**, if you want to create a completely new component use a **component**.

**@Pipe**

**@Pipe** allows you to create your own custom pipes to filter data that is displayed to the user, which canbe very useful. The decorator might look something like this:

**@Pipe**({

name: 'myPipe'

})

which would then allow you to implement it in your templates like this:

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

Now someString would be run through your custom myPipe before the value is output to the user.

**@Injectable**

An **@Injectable** allows you to create a service for a class to use. A common example of a service created using the **@Injectable** decorator, and one we will be using a lot when we get into actually building the apps, is a **Data Service** that is responsible for fetching and saving data. Rather than doing this manually in your classes, you can inject your data service into any number of classes you want, and call helper functions from that **Data Service**. Of course this isn’t all you can do, you can create a service to do anything you like.

An **@Injectable** will often just look like a normal class with the **@Injectable** decorator tacked on at the top:

**@Injectable**()

**export class** DataService {

}

**IMPORTANT:** Remember that just about everything you want to use in Ionic 2 needs to be imported first(we will cover importing in more detail in the next section). In the case of pipes, directives, injectables and components they not only need to be imported, but also declared in your **app.module.ts** file. We will get into the specifics around this when we go through the application examples.

**Summary**

The important thing to remember about decorators is: *there’s not that much to remember*. Decorators are powerful, and you can certainly come up with some complex looking configurations. Your decorators may become complex as you learn more about Ionic 2, but in the beginning, the vast majority of your decorators will probably just look like this:

**@Component**({

selector: 'home-page',

templateUrl: 'home.html'

})

I think a lot of people find decorators oﬀ putting because at a glance they look pretty weird, but they look way scarier than they actually are. In the next lesson we’ll be looking at the decorator’s partner in crime: the class. The class definition is where we will do all the actual work, remember that the decorator just sits at the top and provides a little extra information.

**Class điều khiển**

**(Lesson 5: Classes)**

We know what a class is, but why are they suddenly a thing in Ionic 2 and Angular 2? We’ve touched on this earlier, but **classes** are a new addition to Javascript with the ES6 specification. It is certainly a welcome

change because it is one of the most widely used patterns in programming, and in fact most JavaScript applications were already using classes anyway, ES6 just made it more oﬃcial.

Before ES6 it was common (and I guess it still is since most people are still using ES5) to create a class like structure by using **functions**. This looks a little something like this:

**var** Person = **function** (name, age) {

**this**.name = name;

**this**.age = age;

};

Person.prototype.isOld = **function**() {

**return this**.age > 70;

};

**var** david = **new** Person('david', 72);

**console**.**log**(david.isOld());

It looks a bit diﬀerent, but the end result is basically the same. Since ES6 adds a class keyword we can now use a more ‘normal’ approach.

So let’s take a look at what a class looks like in Ionic 2:

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

**import** { SomePage } from'../pages/some-page/some-page';

**@Component**({

selector: 'page-home',

templateUrl: 'home.html'

})

**export class** HomePage {

**constructor**(**public** nav: NavController) {

}

}

The first thing you will notice is the **import** statements. Anything that is required by the class that you are creating will need to be **imported**. In this case we are importing **Component** from **@angular/core** which allows us to use the **@Component** decorator, and **NavController** from the Ionic library which we can use to control navigation.

We are also importing **SomePage** which is a class of our own creation. The path for this simply follows the directory structure of your project, in this case we have the **SomePage** component defined inside a folder called **pages** which is one level above the current file. The import should link to wherever the **.ts** file is for the class, but it is not necessary to include the **.ts** extension.

Next up we have the decorator, which we use to define the selector (i.e. the name this component will have in our DOM, <page-home></page-home>) and the template.

Once we get past the decorator, we finally arrive at the class itself. Notice though that it is preceded by the **export** keyword, e.g:

**export class** HomePage {

}

The **export** keyword works in tandem with the **import** keyword, so we **export** classes that we want to **import** somewhere else. The last thing we have left to discuss is the constructor. We’ve already talkedabout the role of a constructor in classes in general, and it is no diﬀerent here: the code inside of the constructor is run when the class is instantiated.

There’s a little bit more to it that you need to know though. In Ionic 2 we need to inject any services that we want to use inside of this class into the constructor, which looks like this:

**constructor**(platform: Platform, nav: NavController) {

platform.ready().then(() => {

});

}

In this example we want to use the **Platform** service to detect when the device is ready, so we inject it into the constructor and then we can use it within the constructor.

We don’t need to use **platform** outside of the constructor here, but in most cases you will want to use the services you inject elsewhere in the class as well. So to make the service available to any function within the class, you must **set it as a member variable**. This will look like this:

**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(() =>{});

}

}

**OR** we can use the **public** keyword I mentioned before when using TypeScript to automatically createthese member variable references, like this:

**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(() =>{});

}

}

Also note that any variables declared above the constructor like someMemberVariable is here will also automatically be set up as member variables. So we could access someMemberVariable from anywhere in this class through **this**.someMemberVariable. Any services you need to inject (and possibly set up as member variables) should be added to the constructor, any member variables you want to create for any other purpose should be declared above the constructor. If this concept sounds confusing to you right now it should make a little more sense when we start going through some examples.

Now we can access **platform** from anywhere in this class through **this**.platform. If we did not set up this member variable and tried to access **platform** from our someOtherMethod function, it would not work.

**Creating a Page**

Page’s will usually make up a large portion of your application - for each screen you want to have in your application you will have a separate **Page** defined. Previously, there was a special decorator in Ionic for this, but now they are just regular **@Component**s.

As we’ve already discussed, the class for a page might look something like this:

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

**@Component**({

selector: 'home-page',

templateUrl: 'home.html'

})

**export class** MyPage {

**constructor**() {

}

}

and the template file we are referencing in the decorator might look something like this:

<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**>

The template file makes up what the user will actually see (we’re going to discuss templates in a lot more detail later). The template file and the class work in unison: the class defines what template is to be shown to the user, and the template can make use of data and functions available in the class.

We’ve covered the basic structure of what a **Page** class looks like and what the **constructor** function does, but you can also add other functions that your page can make use of, for example:

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

**@Component**({

selector: 'home-page',

templateUrl: 'home.html'

})

**export class** MyPage {

**constructor**() {

*//this runs immediately*

}

ionViewDidLoad() {

*//this runs once the view has loaded*

}

someMethod(){

*//this only runs when called*

}

someOtherMethod(){

*//this only runs when called*

}

}

You could have your constructor call these other functions or you could even have them triggered by a user clicking a button in the template for example. These additional functions can be added to any class, it’s not just specific to pages. We will cover these concepts in much more detail later, for now we just want to understand the basic structure of the diﬀerent class types and what they do.

**NOTE:** It is considered bad practice to do too much “work” in the **constructor** function. The constructorshould only be used for basic set up, if you want to do more complicated things, you should use something like the ionViewDidLoad hook instead.

**NOTE:** You can auto generate a Page in your project using the commandionic g page MyPage

**Creating a Component**

The code for a generic component looks a lot like the code for a page (remember, a page is a component), just as it will look like just about everything else we create. When creating pages with components we use Ionic’s built in navigation to handle displaying them. A page is a component that will take up the whole screen (i.e. a “view” for the user), but a component will also allow you to create your own custom element that you can insert into your templates as well. Maybe you want to create a custom date picker component to add to your page, or a box that displays random motivational quotes - in both of these cases you would create a custom **Component**.

The class for a generic component will look like this:

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

**@Component**({

selector: 'my-component',

templateUrl: 'my-component.html'

})

**export class** MyComponent {

text: any;

**constructor**() {

**this**.text ='Hello World';

}

}

Really the only diﬀerence with the component here is that we need to supply a **selector** (this is optional for pages). This will be the name of the element that you use to insert the component into your template, i.e:

<my-component></my-component>

Before we get to using it though, let’s also talk about the template for the component. This isn’t really any diﬀerent to how the template for a page works. We’re referencing a file called **my-component.html** which might contain something like this:

<**div**>

{{text}}

</**div**>

Just like with a page, we can reference data that is stored in the class definition (as well as functions). With this template, all our component will do is render:

<**div**>Hello World</**div**>

into the DOM. Which is pretty boring of course, but you can create some pretty interesting, and reusable, stuﬀ with this functionality. So let’s take a quick look at how to now actually use the component in a page.

You will need to import it and add it to your **app.module.ts** file:

@NgModule({

declarations: [

MyApp,

HomePage,

MyComponent

],

imports: [

IonicModule.forRoot(MyApp)

],

bootstrap: [IonicApp],

entryComponents: [

MyApp,

HomePage

],

providers: []

})

**export class** AppModule {}

and then you can just reference it in a pages template like this:

<my-component></my-component>

It’s actually pretty unlikely that you will need to create your own custom components like this in the begin-ning, as Ionic already provides most of what you would need (lists, tabs, buttons, inputs and so on). If you need something that Ionic does not already provide though, then you’ll have to look at creating your own component.

**NOTE:** You can auto generate a Component in your project using the command

ionic g component MyComponent

**Creating a Directive**

As I mentioned before, components and directives are very similar, but in general a component is used to create a completely new element, and a directive is used to modify the behaviour of an existing one.

The class for a custom directive looks like this:

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

**@Directive**({

selector: '[my-directive]'

})

**export class** MyDirective {

**constructor**() {

}

}

In this directive, we also have a **selector** like we did for the component - but it’s slightly diﬀerent. Rather than representing the name of a tag, this can be used as an attribute on an element. You will do this a lot in Ionic 2, on buttons for example:

<**button** ion-**button**>

or on your lists:

<ion-list no-lines>

but in this case we’re creating our own custom directive that we can use on anything, e.g:

<**button** my-directive>

But notice we don’t actually have a template for this directive. Although we usually refer to any feature in our applications as ‘components’, technically speaking a component consists of a class **and** a template (view) - if it does not have a view then it is not a component.

I wanted to just cover the basics here, but I think it’s also useful to know about **ElementRef**. This will give you access to the element that the directive was added to. You can include it in your directive like this:

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

**@Directive**({

selector: '[my-directive]'

})

**export class** MyDirective {

**constructor**(element: ElementRef) {

**this**.element = element;

}

}

You will also need to include this in your **app.module.ts** file just like the custom component.

**NOTE:** You can auto generate a Directive in your project using the command

ionic g directive MyDirective

**Creating a Pipe**

Pipes might seem a little complex at first glance, but they’re actually really easy to implement. They look like this:

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

**@Pipe**({

name: 'myPipe'

})

**@Injectable**()

**export class** MyPipe {

transform(value, args) {

*//do something to 'value'*

**return** value;

}

}

Notice that a pipe is also an **@Injectable**, we will talk about what that is in just a moment. So the idea is that whatever you are passing through the pipe will go to this **transform** function, you do whatever you need to do to the value, and then you **return** the new value back. Now this new value will be rendered out to the screen, rather than the initial value. You can use it in your templates like this:

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

which will run whatever **someValue** is through your custom pipe before displaying it. Once again, make sure you import and reference the pipe in your **app.module.ts** file before using it. “‘

**NOTE:** You can auto generate a Pipe in your project using the commandionic g pipe MyPipe

**Creating an Injectable**

An Injectable allows you to create a service that you can use throughout your application (like an interface between your application and an external or internal data service). Injectables might also be referred to as ‘Providers’. The **@Injectable** that the Ionic CLI automatically generates looks like this:

**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);

});

});

}

}

This code creates a provider called **Data** that loads data from a JSON source (which can either be a local JSON file, an external JSON file or a response from a server). It returns a promise, which will allow the data to be retrieved from the **http** request after it has finished executing. If the data has already been loaded then it just returns the data directly (also through a promise). We’ll go into fetching data with **http** in more depth later, for now we just want to focus on the basics of an injectable.

So if we want to grab the data that this service returns, we would first inject it into wherever we want to use it, i.e:

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

**import** { Data } from '../../providers/data';

**@Component**({

selector: 'home-page',

templateUrl: 'home.html'

})

**export class** MyPage {

**constructor**(**public** dataService: Data){

}

}

add it the the providers array in **app.module.ts**:

providers: [Data]

and then you could make use of any function the injectable provides through **this**.dataService, for example:

**this**.dataService.load().then((data) => {

**console**.**log**(data);

});

Notice that we use then() here because it is a promise that is returned, so we need to wait for the promise to complete before we can access the data. You might extend this provider further so that it also oﬀered a **save** function, i.e:

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

Of course, you can use a provider to do other things besides fetching data - but that’s a very common use case.

**NOTE:** You can auto generate an Injectable in your project using the command

ionic g provider MyProvider