***Giáo trình***

**Lập trình Ứng dụng Di động**

**GS. Phan Viết Hoàng**

**Bài học 10 (Buổi 10)**

**Kết nối server**

# References

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

**Xử lý dữ liệu từ xa**

**(Lesson 11: Fetching Data, Observables and Promises)**

Although some mobile applications are completely self contained (like calculators, soundboards, todo lists, photo apps, flashlight apps), many applications rely on pulling in data from an external source to function. Facebook has to pull in data for news feeds, Instagram the latest photos, weather apps the latest weather forecasts and so on.

In this section we are going to cover how you can pull external data into your Ionic applications. Before we get into the specifics of how to retrieve some data from a server somewhere, I want to cover a little more theory on a few specific things that we are going to be making use of.

**Mapping and Filtering Arrays**

The **map** and **filter** functions are very powerful and allow you to do a lot with arrays. These are not fancy new ES6 or Angular 2 features either, they’ve been a part of JavaScript for a while now.

To put it simply, **map** takes every value in an array, runs the value through some function which may change the value, and then places it into a new array. It **maps** every value in an array into a new array. To give you an example of where this might be useful, you might have an array of filenames like this:

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

You could then map those values into a new array that contains the full path to the files:

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

Doing that might look something like this:

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

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

});

So we supply the **map** with a function that returns the modified value. The function that we provide will have each value passed in as a parameter.

A **filter** is very similar to a **map**, but instead of mapping each value to a new array, it only adds values that meet a certain criteria to the new array. Let’s use the same example as before, but this time we want to return an array that only contains .png files. To do that, we would use filter like this:

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

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

});

Suppose we still want to have the full path as well though. Fortunately, we can quite easily chain **filter** and **map** like this:

**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;

});

So now we are first filtering out the results we don’t want, and then mapping them to a new array with the full file path. The result will be an array containing the full paths of only the two .png files. I’m going to leave it there for now, but when we get into an example of how to fetch data soon it will become very clear why it was worth explaining how these work.

**Observables and Promises**

If you’ve used Ionic 1 or have a reasonably strong background in Javascript then you would probably be familiar with **Promises**, but far fewer people are familiar with **Observables**. Observables are one of the core new features included Angular 2 (provided by RxJS) so it is important to understand what they are and how they are diﬀerent to Promises (they do look and behave *very* similar).

Before we get into Observables, let’s cover what a Promise is at a very high level. Promises come into play when we are dealing with **asynchronous** code, which means that the code is not executed one line after another. In the case of making a HTTP request for data, we need to wait for that data to be returned, and since it might take 1-10 seconds for it to be returned we don’t want to pause our entire application whilst we wait. We want our application to keep running and accepting user input, and when the data from the HTTP request becomes available to us, *then* we do something with it.

A Promise handles this situation, and if you’re familiar with callbacks it’s basically the same idea, just a little nicer. Let’s say we have a method called getFromSlowServer() that returns a promise, we might use it like this:

getFromSlowServer().then((data) => {

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

});

We call the then method which a Promise provides, which basically says “Once you have the data from the server, do this with it”. In this case we are passing the data returned into a function where we log it out to the console. So our application will go about doing whatever else it has to do and when the data is available it will execute the code above. You could think of it like being at work and writing some report, you need some additional information so you ask your assistant to go find it for you, but you don’t just sit there and wait for the assistant to get back - you keep writing your report and when the assistant returns then you use the information.

We understand what a Promise is now, so what’s an Observable and what does it do that Promises don’t?

An Observable serves the exact same purpose as a Promise, but it does some extra stuﬀ too. The main difference between a Promise and an Observable is that a Promise returns a single result, but **an Observable** **is a stream than can emit more than one value over time**. It might be easier to think of Observables asstreams, because they are, they are just called Observables because the stream is observable (as in, we can detect values that are emitted from the stream).

An Observable looks a lot like a Promise, but instead of using the then method we use the subscribe method. Since a Promise only returns a single value, it makes sense to have that value returned and then do something. As I mentioned, an Observable is a stream that can emit multiple values, so it makes sense to subscribe to it (like [your favourite YouTube channel](https://www.youtube.com/c/JoshuaMorony)), and run some code every time a value is emitted. It might look something like this:

someObservable.subscribe((result) => {

**console**.**log**(result);

});

It is obvious that our program would need to wait for data to be returned when making a HTTP request, and thus Promises and Observables would be useful. It’s not the only instance of where you will need to program asynchronously though. There are some less obvious situtations like fetching locally stored data, or even getting a photo from the user’s camera, where you would also need to wait for the operation to finish before using the data.

If you want to go more indepth into everything we’ve discussed above, I highly recommend [this interactive](http://reactivex.io/learnrx/) [tutorial](http://reactivex.io/learnrx/). It introduces RxJS which includes Observables, but also builds up a solid foundation of how to use **map**, **filter** and other functions. If you’d also like to dive into some more specifics about how an Observable diﬀers from a Promise, I highly recommend [this egghead.io video](https://egghead.io/lessons/rxjs-rxjs-observables-vs-promises).

**Using Http to Fetch Data from a Server**

Ok, you should be armed with all the theory you need now - let’s get into an example. We’re going to use the Reddit API to demonstrate here because it is publicly accessible and very easy to use. If you’ve purchased one of the packages for this book that includes the Giflist application then we will be exploring this in a lot more detail later.

You can create a JSON feed of posts from subreddits simply by visiting a URL in the following format:

<https://www.reddit.com/r/gifs/top/.json?limit=10&sort=hot>

If you click on that link, you will see a JSON feed containing 10 submissions from the **gifs** subreddit, sorted by the hot filter. If you’re not familiar with JSON, I would recommend reading up on it [here](https://msdn.microsoft.com/en-us/library/bb299886.aspx) – but essentially it stands for JavaScript Object Notation and is a great way to transmit data because it is very readable to humans, and is also easily parsed by computers. If you’ve ever created a JavaScript object like this:

**var** myObject = {

name: 'bob',

age: '43',

hair: 'purple'

};

then you should be able to read a JSON feed pretty easily once you tidy it up a little. But how do we get it into our Ionic 2 application?

The answer is to use the **Http** service which is provided by Angular 2, and allows you to make HTTP requests. If you’re not familiar with what a HTTP request is, basically every time your browser tries to load anything (a document, image, a file etc.) it sends a HTTP request to do that. So we can make a HTTP request to a page that spits out some JSON data, and pull that into our application.

First we need to set up the **Http** service, so let’s take a look at a test page that has that service imported and injected into the constructor:

**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) {

}

}

Since we have injected the Http service into our constructor and made it available through **this**.http by using **public**, we can now make use of it anywhere in this class. Also note that we are importing the map operator from the RxJS library. As I mentioned before map is a function that is provided by default on arrays - so why would we need to import it from some weird library? It’s because the Http service doesn’t return an array, it returns an **Observable**. The RxJS library makes the map function available for us on Observables, but we need to import it first.

Now let’s take a look at how we might make a request to a reddit URL:

**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

=> res.json()).subscribe(data => {

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

});

}

}

The first part of the call returns us an Observable. Then we make use of the **map** function, what we’re doing here is taking the plain text JSON response (which is just a string) and converting it into a JavaScript object by calling the json()function. This makes it much more friendly for us to play with.

**IMPORTANT:** Remember, Http requests are asynchronous. This means that your code will continue exe-cuting whilst the data is being fetched, which would take anywhere from a few milliseconds, to 10 seconds, to never. So it’s important that your application is designed to deal with this. To give you an example, if you were to run the following code:

**this**.posts = null;

**this**.http.get('https://www.reddit.com/r/gifs/top/.json?limit=2&sort=hot')

.map(res => res.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 => res.json()).subscribe(data => { **this**.posts = data.data.children;

**console**.**log**(**this**.posts);

});

You would see the posts output to the console, because everything inside of the **subscribe** function will only run once the data has been returned.

Getting back to our example, after we map the response we chain a subscribe call which allows us to do something with the data that is emitted from the stream (Observable). As I mentioned above an Observable is useful because we can listen for multiple values over time… but why use it here? The Http call is only ever going to return one result, why doesn’t it just use a Promise instead of an Observable and save everyone the confusion?

The reason for a bit of favouritism of Observables over Promises is that an Observable can do everything a Promise can, it’s technically better behind the scenes, and it can do extra fancy things that Promises can’t. We could set up an **interval** for example so that the Http call fires every 5 or 10 seconds, we can easily set up **debouncing** which ensures that a request is not fired oﬀ too frequently (and subsequently making a ton of requests to a server), Observables can cancel old “in flight” requests if a new request is made before the result of the old request is returned and a whole bunch of other things.

Take this example from the Giflist application:

**this**.subredditControl.valueChanges.debounceTime(1000)

.distinctUntilChanged().subscribe(subreddit => {

**this**.subreddit = subreddit;

**if**(**this**.subreddit !=‘’){

**this**.changeSubreddit();

}

});

In this case subredditControl is an **Observable**. The value of this can be controlled by the user using an input in the application. It’s set up though so that the code inside the **subscribe** call will only run if there has been no change for more than 1 second (by using debounceTime) and it will also only run when a distinct value is supplied. I think this example shows well how a whole bunch of weird and useful stuﬀ can be chained before the **subscribe** call to do a lot of useful things.

If you’re looking at the example above and thinking “Wow… Ionic 2 is way too hard and confusing, take me back to Ionic 1!” then don’t. This is a pretty advanced example using Observables to demonstrate a point, Ionic 2 input handling and two way data binding is just as easy as Ionic 1.

Observables are a huge topic, so there is a ton to learn. Don’t feel intimidated if you don’t really have much of an idea of what’s going on though. Having a better understanding of Observables will help you when creating Ionic applications, but they are a reasonably small part of Ionic (well, they are a big part but you won’t really have to deal with them much) and you can get by just fine by having just a basic understanding.

**Fetching Data from your Own Server**

We know how to pull in data using a JSON feed like the one provided by reddit, but what if you want to pull in your own data? How can you go about setting up your own JSON feed?

Going into the detail of how to set up your own API is a bit beyond what I wanted to achieve with this lesson, but I would like to give you a high level overview of how it’s done. Basically:

1. Make a request from your Ionic application to a URL on your server
2. Fetch the data using whatever server side language you prefer
3. Output the required data to the page in JSON format

I’ll quickly walk you through the steps of how you might implement a simple API with PHP, but you could use whatever language you want - as long as you can output some JSON to the browser.

1. Create a file called feed.php that is accessible at http:*//www.mywebsite.com/api/feed.php*
2. Retrieve the data. In this case I’m doing that by querying a MySQL database but the data can come from anywhere:

$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. Output the JSON encoded data to the browser:

echo($result);

1. Use http:*//www.mywebsite.com/api/feed.php* in your http.get() call in your application

As I mentioned you can use whatever language and whatever data storage mechanism you like to do this.

Just grab whatever data you need, get it in JSON format, and then output it to the browser.

This should give you a pretty reasonable overview of how to fetch remote data using Ionic 2 and the Http service. The syntax and concepts might be a little tricky to get your head around at first, but once you’ve got the basics working there’s not really much more you need to know. Your data might get more complex and you might want to perform some fancier operations on it or display it in a diﬀerent way, but the basic idea will remain the same.