***Giáo trình***

**Lập trình Ứng dụng Di động**

**GS. Phan Viết Hoàng**

**Bài học 5 (Buổi 5)**

**Thiết kế giao diện**

# References

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

**Thêm control vào giao diện**

**(Lesson 6: Templates)**

Templates, I think, are one of the most fun bits of Ionic 2. It’s where the power of the framework really shines. Take this code for example:

<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-circle"></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**>

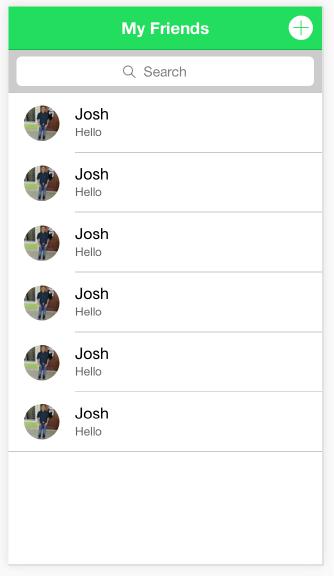
<**p**>{{item.description}}</**p**>

</ion-item>

</ion-list>

</ion-**content**>

with no additional styling, the code above would look like this right out of the box:



It doesn’t look amazing, but we already have a pretty complex layout set up with just a few lines of code, throw a bit of custom styling in and we’d have a pretty sleek interface. We’re going to go through diﬀerent aspects of creating templates in Ionic 2 more thoroughly in just a moment, but I wanted to give you a sense of what a full template for a page might look like, and also how easy it is to use the components provided by Ionic.

There’s a lot more to know about template syntax in Ionic 2, and there’s been a few significant changes from what you might have been used to in Ionic 1 - so we’re going to dive into templates in a lot of detail.

We’ll also be covering quite a few other topics after this, but this is the last of what I would consider to be the “core” knowledge required to get started with Ionic 2 - once you’ve got the basics of classes and templates down you can start jumping into building some stuﬀ. So, strap in - we’re going to start oﬀ with a little theory then get into some practical examples.

**The \* Syntax**

Perhaps one of the most confusing things about the new template syntax in Ionic 2 is this little guy: \*.

You’ll often come across code that looks like this:

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

or

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

and so on. In Angular 2 the \* syntax is used for structural directives which are a shortcut for creating an embedded template, so if we use \*ngIf as an example, the above could be expanded to:

<template [ngIf]="someBoolean">

<**p**></**p**>

</template>

The reason for using templates is that Angular 2 treats templates as chunks of the DOM that can be dynamically manipulated. So in the case of \*ngIf we don’t want to just literally render out:

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

to the DOM. We want to render out:

<**p**></**p**>

if someBoolean is true, and nothing if it is false. Similarly, if we were to use \*ngFor we don’t want to literally render out:

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

we want to render out paragraph tags stamped with the information for each particular item:

<**p**>Bananas</**p**>

<**p**>More Bananas</**p**>

<**p**>Pancakes</**p**>

So we need to use <template> to allow for this functionality, but writing out templates manually is a lot of work, and the \* syntax just makes this a lot easier.

Now that we’ve got that out of the way, let’s jump into some specifics of how to use structural directives like \*ngIf and \*ngFor.

**Looping**

Quite often you will want to loop over a bunch of items - when you have a list of articles and you want to render all of the titles into a list for example. We can use the **ngFor** directive which is supplied by Angular 2 to achieve this - it looks something like this:

<ion-list>

<ion-item \*ngFor="let article of articles" (click)="viewArticle(article)">

{{article.title}}

</ion-item>

</ion-list>

In this example, we create an <ion-list> and then for every **article** we have in our **articles** array we add an <ion-item>. I mentioned before (in the basics section) the use of **let** to create a local variable and we are using that here. This allows us to access whatever **article** the loop is currently up to, and we are using that to grab the title of the current article and render it in the list, and also to pass it into the viewArticle function that is triggered when the item is clicked.

By passing a reference to the current article to the viewArticle function we would then be able to do something like trigger a new page with the specifics of that article on it.

**Conditionals**

Sometimes you will want to display certain sections of the template only when certain conditions are met, and there’s a few ways to do this.

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

ngIf will render the node it is attached to only if the expression evaluates to be true. So in this case, if someBoolean is true, it will be added to the DOM, if it is false then it will not be added to the DOM.

ngIf is great for boolean - true or false - scenarios, but sometimes you will want to do multiple diﬀerent things based on a value. In that case you can use 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**>

In this example we are checking the value of paragraphNumber with ngSwitch. Whichever ngSwitchCase statement the value matches will be the DOM element that will be rendered, and if none match the ngSwitchDefault element will be used.

Another method to display or hide certain elements based on a condition is to use the hidden property.

For example:

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

In this example, if the hideAvatar expression evaluates to be **true** the element will be hidden, but if it is **false** then it will be rendered. Using this method, you would need to have ahideAvatarmember variablein your class definition which you could toggle to hide and show these elements. This is called a property binding, by using the square brackets we are able to reference values from the class, without the square brackets it would treat “hideAvatar” literally.

As well as conditionally displaying an entire element, you could also attach diﬀerent classes to an element based on a condition, for example:

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

This is a similar concept to the [hidden] method above, but instead of showing and hiding the element based on a condition, it will add a class you have defined in your CSS based on a condition. This can come in really handy, for example you might want to use it to style items that have already been read by the user a diﬀerent colour.

**Ionic 2 Template Components**

Everything we’ve covered above is general Angular 2 stuﬀ, there’s nothing specific to Ionic there (except for the <ion-list> and <ion-item> elements we used). You will be using this syntax a lot throughout your templates, along with some Ionic specific components. We’re going to go through some of the Ionic specific stuﬀ now, starting with the basic layout of an Ionic 2 page template:

<ion-header>

<ion-navbar>

<ion-**title**>

Ionic Blank

</ion-**title**>

</ion-navbar>

</ion-header>

<ion-**content** padding>

</ion-**content**>

This is the automatically generated code you will get for your template if you use the blank layout. All of the elements here are **components** just like the components we can create, except these are provided by default by Ionic. There’s two important components here that are used in just about every template and they are <ion-navbar> and <ion-content>.

The <ion-content> element is simply used to hold the main content of the page, and allows for scrolling.

The more interesting of the two is <ion-navbar>. This is what adds the header bar to the top of the page, where you can place the title of the page as well as buttons on the left or right of the navigation bar. It’s not purely an aesthetic thing though, it also has a lot of inbuilt smarts for navigation. If you were to **push** a new page (a concept we will cover in detail later), then a back button will automatically be added tothe <ion-navbar> which will automatically allow the user to navigate back to the previous page, rather than you having to handle that manually.

All of the above covers the basic template syntax you will see in a lot of your Ionic 2 pages, the rest is basically just dropping in and configuring various components that Ionic 2 oﬀers (or if you’re feeling adventurous, your own custom components).

Now let’s take a look at how to implement a few of Ionic’s components into our templates. We’re not going to be covering anywhere near all of them because there is so many, I just want to give you a taste. For a full list of all the available components, take a look at the [Ionic 2 documentation](http://ionicframework.com/docs/v2/components/#overview).

**Lists**

Lists are one of the most used components in mobile applications, and they provide an interesting chal-lenge. That smooth scrolling you get when you swipe a list on native applications, with smooth acceleration and deceleration, and it all just *feels right* - well that’s *really* hard to replicate. Fortunately, you don’t have to worry about it because Ionic 2 does all the hard stuﬀ for you, and using a list is as simple as adding the following to your template:

<ion-list>

<ion-item>Item 1</ion-item>

<ion-item>Item 2</ion-item>

<ion-item>Item 3</ion-item>

</ion-list>

or if you wanted to dynamically create your list for a bunch of items defined in your class:

<ion-list>

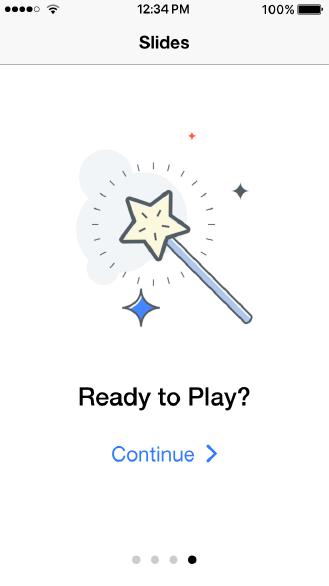
<ion-item \*ngFor="let item of items" (click)="itemSelected(item)">

{{item.title}}

</ion-item>

</ion-list>

**Slides**

Slides are another common element used in mobile applications, slides look like this

where you have multiple diﬀerent images or pages to show and the user can cycle through them by swiping left or right. Just like lists in Ionic 2, creating slides is really easy as well:

<ion-slides >

<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>

A container of <ion-slides> is used, and then each individual slide is defined with <ion-slide>. It is also possible to supply some options to define the behaviour of the slides (I’ll give you a more complete example later).

**Input**

In Ionic 2, rather than using <input> tags for user input you use the Ionic equivalent which is <ion-input>. Just like a normal <input> you can specify a type depending on what sort of data you are capturing, but by using the Ionic versions we will be taking advantage of the custom inputs Ionic has designed for mobile.

<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>

As well as <ion-input> specifically, you will also find that Ionic provides other input elements like <ion-select>, <ion-radio>, <ion-checkbox> and <ion-toggle>.

**Grid**

The Grid is a very powerful component, and you can use it to create complex layouts. If you’re familiar with CSS frameworks like Bootstrap, then it is a very similar concept. When placing components into your templates, in general things just display one after the other, but with the Grid you can come up with just about any layout you can imagine.

It works by positioning elements on the page based on rows and columns. Rows display underneath each other, and columns (which are placed inside of rows) display side by side. For example:

<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>

This will create a layout with two rows, the top row will have two columns and the bottom row will have three columns. By default everything will be evenly spaced, but you can also specify how wide columns should be if you like:

<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>

This will create a single row with 5 diﬀerent columns of varying widths (you will probably want to make sure your column widths add up to 100!).

For a full list of available widths take a look at [the documentation](http://ionicframework.com/docs/v2/components/#grid).

**Icons**

Icons are heavily used in most applications today, they are great because they allow you to communicate what something does rather than relying on text. It’s good for usability (most of the time) and looks way better than using a button that says something like “Add Item”.

Ionic provides a ton of icons that you can use out of the box, like:

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

You just have to specify the name of the icon you want to use. They even have variations of the same icons so that they display diﬀerently between iOS and Android to better match the style of the platform. For a full list of available icons you can [go here](http://ionicframework.com/docs/v2/ionicons/).

There’s a ton more default components, and a lot more to know about even the ones I’ve mentioned here, so make sure you have a look through [the documentation](http://ionicframework.com/docs/v2/components/#overview) to familiarise yourself with what’s available.

**Thêm form vào giao diện**

**(Lesson 9: User Input)**

Not all mobile applications require user input, but many do. At some point, you’re going to want to collect some data from your users. That might be some text for a status update, their name and shipping address, a search term, a title for their todo list item or anything else.

Whatever the data is, the user is going to be entering it into one of the templates in your application. To give you an example, in Ionic 2 we could create a form in our template with the following code:

<ion-list>

<ion-item>

<ion-**label**>Username</ion-**label**>

<ion-**input type**="text"></ion-**input**>

</ion-item>

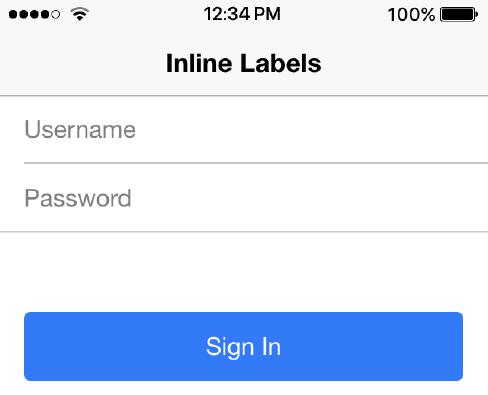
<ion-item>

<ion-**label**>Password</ion-**label**>

<ion-**input type**="password"></ion-**input**>

</ion-item>

</ion-list>

which would produce a simple login form that looks like this

This would allow the user to enter some information into these input fields. However, we need to know how to get the data that is being entered into our .html template and make use of it in our .ts class. In this lesson we are going to discuss a couple of diﬀerent ways you can do that.

**Two Way Data Binding**

This concept will be very familiar to you if you’ve previously used Ionic 1, if not then don’t worry because it’s pretty straightforward concept. Two way data binding essentially links a value of an input field in the template, to the value of a variable in the class. Take the following example:

**Template:**

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

**Class:**

myValue: string;

**constructor**(){

}

If we changed the value of the input in the template, then the **this**.myValue variable in the class will be updated to reflect that. If we change the value of **this**.myValue in the class, then the input in the template will be updated to reflect that. By using ngModel the two values are tied together, if one changes, the other changes.

Let’s say you also had a submit button in your template:

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

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

When the user clicks the button we want to log the value they entered in the input to the console. Since the button calls the logValue function when it is clicked, we could add that to our class:

logValue(){

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

}

This function will grab whatever the current value of the input is and log it out to the screen. Rather than logging it out to the screen, you could also do something useful with it. This can be a convenient way to handle input, because we don’t need to worry about passing the values through a function, we can just grab the current values whenever we need.

It becomes a bit cumbersome when we have a lot of inputs though, so it’s not always the perfect solution.

When dealing with more complex forms, we also have another option, which we will discuss now.

**Form Builder**

Form Builder is a service provided by Angular 2, which makes handling forms a lot easier. There’s quite a lot Form Builder can do but at its simplest it allows you to manage multiple input fields at once and also provides an easy way to validate user input (i.e. to check if they actually did enter a valid email address).

To use Form Builder it needs to be imported (along with FormGroup) and injected into your constructor, e.g:

**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) {

}

}

Notice that **Validators** are also being imported here, which are what allow you to validate user input with Form Builder. Let’s cover a really quick example of how you can use Form Builder to build a form. The most important diﬀerence with this method is that your inputs will have to be surrounded by a <form> tag with the formGroup property defined:

<**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**>

You will also notice in the example above that we have defined the formControlName attribute on all of our inputs, this is how we will identify them with Form Builder in just a moment. Of course we need a way to submit the form, so we’ve added a submit button and have also added a (submit) listener on the form which calls the saveForm function. We will define that function in a moment, but for any of this to work we first need to initialise the form in the constructor function for the page. This will look something like this:

**this**.myForm = formBuilder.group({

field1: [''],

field2: [''],

field3: ['']

});

We simply supply all of the fields that are in the form (using the formControlName names we gave them) and provide the fields with an initial value (which we have left blank in this case). You can also supply a second value to each field to define a Validator if you like, e.g:

**this**.myForm = formBuilder.group({

field1: ['', Validators.required],

field2: ['', Validators.required],

field3: ['']

});

Also note that the variable **this**.myForm has to be the same as the value we supply for [formGroup] in the template. Now let’s look at that saveForm function.

saveForm(event){

event.preventDefault();

**console**.**log**(**this**.myForm.value);

}

We pass through the submit event and then call preventDefault so that the default action for submitting a form doesn’t occur, we just want to handle it ourselves with this function. To grab the details that the user entered into the form we can simply use **this**.myForm.value which will contain all the values that the user entered.

Setting up forms using Form Builder is a little more complex, but it’s much more powerful and worth the eﬀort for more complex forms. For more simple requirements, using [(ngModel)] is fine in most cases.

**Làm đẹp giao diện**

**(Lesson 7: Styling & Theming)**

**Introduction to Theming in Ionic 2**

When styling an Ionic 2 application, there is nothing inherently diﬀerent or special about it – it’s no diﬀerent than the way you would style a normal website. I often see questions like:

“Can I create [insert UI element / interface] in Ionic?”

and the answer is generally **yes**. Could you do it on a normal webpage? If you can then you can do it in Ionic as well.

A lot of people may be used to just editing CSS files to change styles, but there is some added complexity with Ionic, which is primarily due to the fact that it uses **SASS**. Again, SASS isn’t specific to Ionic or mobile web app development – it can also be used on any normal website – but many people may not be as familiar with SASS as they are with plain old CSS.

If you’re not already familiar, **.scss** is the file type for **SASS** or **Syntactically Awesome Style Sheets**. If this is new to you, you should read more about what SASS is and what it does [here](http://sass-lang.com/). For those of you short on time, what you put in your **.scss** files is exactly the same as what you would put in **.css** files, you can just do a bunch of extra cool stuﬀ as well like define variables that can be reused in multiple areas. These **.scss** files are then compiled into normal **.css** files (it’s basically the same concept we use in Ionic 2, where we code using all the fancy new ES6 features, but that is then transpiled into ES5 which is actually supported by browsers now).

When theming your application, you’re mainly going to be editing your **.html** templates and **.scss** stylesheets – you will **NEVER** edit any .css files directly. The .css files are generated from the .scss files, so if you make any changes to the .css file it’s just going to get overwritten.

If you take a look at the files generated when you create a new Ionic 2 project, you will see some **.scss** files inside of your **theme** folder, so let’s quickly run through what their purpose is.

* **src/app/app.scss** is used to declare any styles that will be used globally throughout the application
* **theme/variables.scss** is used to modify the apps shared variables. Here you can edit the default

values for things like $colors which sets up the default colours for the application, as well as $list-background-color, $checkbox-ios-background-color-on and so on. The gen-eral idea is that Ionic uses the variables defined in this file to determine styling for a lot of components, so it can be a great place to make quick changes. For a list of all the variables that you can overwrite, take a look at [this page](http://ionicframework.com/docs/v2/theming/overriding-ionic-variables/)

On top of these **.scss** files inside of your **theme** folder, you will also have one for each component you create (or at least you should). To refresh your memory, most components you create in Ionic 2 will look like this:

**my-component**

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

We have the class definition in the .ts file, the template in the .html file and any styles for the component in the .scss file. Although it’s not strictly required, you should always create the .scss file for any components that have styling, rather than just defining the style in the **app.core.scss** file. If you use the auto generate commands the Ionic CLI provides then the .scss file will be created automatically for you anyway.

Why? Well you *could* just put all of your styles in the **app.core.scss** file and everything would work exactly the same, but there’s two major benefits to splitting your styles up in the way I described above:

**Organisation** – Splitting your code up in this way will keep the size of your files down, making it a lot easierto maintain. Since all of the styles for a particular component can be found in that components .scss file, you’ll never have to search around much.

**Modularity** – one of the main reasons for the move to this component style architecture in Angular 2 andIonic 2 is modularity. Before, code would be very intertwined and hard to separate and reuse. Now, almost all the code required for a particular feature is contained within its own folder, and it could easily be reused and dropped into other projects.

Now that we’ve gone over the theory, let’s look at how to actually start styling our Ionic 2 applications.

**Methods for Theming an Ionic 2 Application**

I’m going to cover a few diﬀerent ways you can alter the styles in your application. It may seem a little unclear what way to do things, because in a lot of cases you could achieve the same thing multiple diﬀerent ways. In general, you should try to achieve what you want to do without creating custom styles (which we will cover last here). Instead you should first try using the pre-defined attributes or overriding SASS variables. If it can not be done any other way, then look into creating your own custom styles. Don’t worry too much though, just try to keep things as simple as you can.

**1. Attributes**

One of the easiest ways to change the style of your application is to simply add an attribute to the element you’re using. As I mentioned above, SASS is used to define some colours, and these are:

* primary
* secondary
* danger
* light
* dark
* favorite

which you can see defined in the **src/theme/variables.scss** file:

$colors: (

primary: #387ef5,

secondary: #32db64,

danger: #f53d3d,

light: #f4f4f4,

dark: #222,

favorite: #69BB7B

);

As you can see above, Ionic provides some defaults for what these colours are, but you can also override each of these to be any colours you want. So if you add the **primary** attribute to most elements it will turn blue, or if you add the **danger** attribute it will be a red colour. But if you modified these then **primary** could make things purple and **danger** could make things pink.

To give you an example, if I wanted to use the **secondary** colour on a button I could do this:

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

or if I wanted to use the secondary colour on a the nav bar I could do this:

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

Keep in mind that these attributes aren’t limited to just changing the colour of elements, some attributes will also change things like the position:

<ion-navbar **color**="secondary">

<ion-buttons end>

<**button** ion-**button color**="primary">I'm a primary coloured button in the end position of the nav bar</**button**>

</ion-buttons>

</ion-navbar>

The example above uses the end attribute to decide where the buttons should appear. Also notice that we use the ion-button attribute with the button here, this let’s Ionic know that we want to use the Ionic styling for the button. We could also control whether or not a list should have borders:

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

or even whether a list item should display an arrow to indicate that it can be tapped:

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

There’s a bunch more of these attributes, so make sure to poke around the documentation when you are using Ionics in built components. The no-lines attribute is a real easy way to remove lines from a list, but if you didn’t know this attribute existed (which is quite possible) then you’d likely end up creating your own custom styles unneccesarily. This is why I recommend trying to do things with attributes first if you can, because you could save yourself a lot of eﬀort.

**2. SASS Variables**

The next method you can use to control the style of your application is to change the default SASS variables (like editing the $colors we talked about above). These are really handy because it allows you to make app wide style changes to specific things. I touched on SASS variables before, but basically in your .scss files you can do something like this:

$my-variable: red;

and then you could reference $my-variable anywhere in the .scss file. So for example if you wanted to make the background colour on 20 diﬀerent elements red, rather than doing:

background-color: red;

for all of them, you could instead do this:

background-color: $my-variable;

The benefit of this is that now if you wanted to change the background color from red to green, all you have to do is edit that one variable – not every single class you have created. This is why you’ll find that variables are named in the manner of **primary** and **danger** rather than specifically **blue** and **red**. There may come a time when you want to change your primary colour to be purple, but if you give variables specific names like $my-blue-color and you change it to be purple it’s going to make your code pretty confusing.

You probably won’t be creating many of your own variables, but Ionic defines and uses a bunch of these variables, and you can easily overwrite them to be something else. Let’s take a look at a few:

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

You can look at the [documentation](http://ionicframework.com/docs/v2/theming/overriding-ionic-variables/) for more information on these and what they default to, but it’s pretty clear by their name what they do. As you can see by the last example there, they even get very specific.

Editing these variables is really simple, just open **variables.scss** and insert your own definitions. Here’s an example **variables.scss** from one of the applications in this book:

$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;

In this example some of the default colours have been changed, and some overrides for specific styles on both iOS and Android are provided.

Notice the use of **md** here, this stands for material design and is used for Android. Ionic 2 seamlessly adapts to the conventions of the platform it is running on with little to no style changes required from you – for Android this means material design is used.

The great thing about editing these default SASS variables is that you can, with one change, make all the changes necessary everywhere in the app. Some variables use the values of other variables, so if you wanted to just do this manually with CSS you would probably need to make a lot of edits to get the eﬀect you wanted.

**3. Configuration**

Another convenient way to change the styling of your application is through the **Config** object that you can provide to **IonicModule** in **app.module.ts**.

In general this is used for setting app wide defaults like the placement of buttons and tabs, the style of icons to be used, transitions and so on. Usually it’s best to leave these unaltered unless you have a specific reason for changing it, since Ionic will adapt to the conventions of the platform it is running on automatically - messing with the config could break this.

Sometimes you will want to force things to be a certain way though, and the **Config** can be a good way to do that. Here’s an example of what it might look like:

IonicModule.forRoot(MyApp, {

backButtonText: 'Go Back',

iconMode: 'ios',

modalEnter: 'modal-slide-in',

modalLeave: 'modal-slide-out',

tabbarPlacement: 'bottom',

pageTransition: 'ios'

})

and if you wanted to force iOS to use Material Design you could set the mode using the Config options:

IonicModule.forRoot(MyApp, {

mode: 'md'

})

Again, I’d stress against doing something like this unless you have a good reason. You might like and be used to material design if you’re an Android user, but your users on iOS (and vice versa) will not have the same view as you. With that in mind, the Config also allows you to configure things specifically for specific platforms like this:

IonicModule.forRoot(MyApp, {

tabbarPlacement: 'bottom',

platforms: {

ios: {

tabbarPlacement: 'top',

}

}

})

For more information on the Config object, take a look at [the documentation](http://ionicframework.com/docs/v2/api/config/Config/).

**4. Custom Styles**

Before we talked about using attributes to change the colours of elements. Given that you can override these attributes to whatever you like, it’s a good approach to set the primary, secondary, danger etc. variables to match the colour palette of your design, and then use those to set the styles of elements, rather than defining custom CSS classes.

But, sometimes there will come a time where you need to define some plain old CSS classes to achieve what you want. You can either define these custom classes in **src/app/app.scss** if the class will be used throughout the application, or in an individual components .scss file if it is only going to be used for one component.

Of course, you can also define custom styles on the element directly by using the style tag, but make sure you use this sparingly.

As you can see, there’s a few diﬀerent ways you can change the styling of your Ionic 2 applications. In general, it’s best to do as little as possible to achieve what you need. Try to achieve as much as you can with attributes and SASS variables, because it will make your life easier.

As I mentioned before, Ionic seamlessly adapts to the UI conventions of both iOS and Android, so the more “hacky” or “brute force” your solution for styling is, the greater chance you have of breaking this behaviour.

**Kết nối trang**

**(Lesson 8 Tem: Navigation)**

If you come from an Ionic 1 or Angular 1 background, then you would be used to handling navigation through routing with URLs, states and so on. The focus in Ionic 2 though is using a navigation stack, which involves **pushing** views onto the **navigation stack** and **popping** them oﬀ. Before we get into the specifics of how to implement this style of navigation in Ionic 2, let’s try to get a conceptual understanding of how it works first.

**Pushing and Popping**

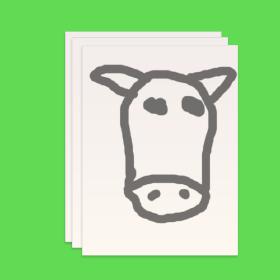
Imagine your **root page** is a piece of paper that has a picture of a cat on it, and you put that piece of paper on a table. It is the only piece of paper currently on the table and you are looking down on it from above. Since it is the only piece of paper on the table right now, of course you can see the picture of the cat:



Now let’s say you want to look at a diﬀerent piece of paper (i.e. go to a diﬀerent page), to do that you can **push** it onto the stack of papers you have. Let’s say this one is a picture of a dog, you take that piece ofpaper and place it over the top of the picture of the cat:



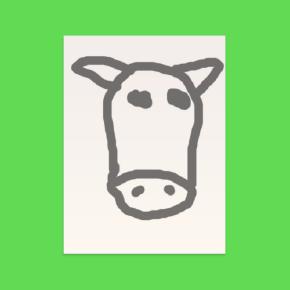
The cat is still there, but we can’t see it anymore because it is behind the dog. Let’s take it even further and say that now you want to **push** another piece of paper, a cow, it would now look like this:



Both the cat and the dog are still there, but the cow is on top so that is what we see. Now let’s reverse things a bit. Since all of the pieces of paper are stacked in the order they were added we can easily cycle back through them by **popping**. If you want to go back to the picture of the dog you can **pop** the stack of papers, removing the piece of paper that is currently on top (the cow). If you want to go back to the picture of the cat you can **pop** the stack of papers once more to remove the piece of paper that is now on top (the dog). Now we’re back to where we started.

I’m sure you can see how this style of navigation is convenient for maintaining history and it makes a lot of sense when navigating to child views, but it doesn’t always make sense to **push** or **pop**. Sometimes you will want to go to another page without the ability to go directly back to the page that triggered the change (a login screen that leads to the main app for example, or even just diﬀerent sections of an app available through a menu).

In this case, we could change the root page which, given our pieces of paper on the table analogy, is like disregarding the other stack of papers we have and just focusing on a new piece of paper on the table:



In the example above, I’ve set the cow page as the root page, so rather than being on top of the other pages, it’s all by itself.

At first, it may be hard to understand whether you should set the **root page** to navigate to a diﬀerent page or push the view. In general, if the view you want to switch to is a child of the current view, or if you want the ability to navigate back to the previous view from the new view, you should **push**. For example, if I was viewing a list of artists and tapped on one I would want to **push** the details page for that artist. If I was going through a multi-page form and clicked ‘Next’ to go to page 2 of the form, I would want to push that second page.

If the view you are switching to is not a child of the current view, or it is a diﬀerent section of the application, then you should instead change the **root page**. For example, if you have a login screen that leads to the main application you should change the root page to be your main logged in view once the user has successfully authenticated. If you have a side menu with the options **Dashboard**, **Shop**, **About** and **Contact** you should set the **root page** to whichever of these the user selects.

Keep in mind that **the root page is diﬀerent to the root component**, typically the root component (which is defined in **app.component.ts**) will declare what the root page is – the root page can be changed throughout the application, the root component can not.

**Basic Navigation in Ionic 2**

Ok, we’ve gone through the theory so now we’re going to get into a more practical Ionic 2 example and look at how to **push**, **pop**, set the **root page** and even how to **pass data between pages**.

An important part of all this is the **NavController** which is provided by Ionic. You will often see this imported in Ionic 2 applications:

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

**@Component**({

selector: 'home-page',

templateUrl: 'home.html'

})

**export class** HomePage {

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

}

}

We inject the **NavController** and a reference to it is created so that we can use it anywhere within the class. As you might have been able to guess, the NavController helps us control navigation - so let’s take a look at how to do just that by **pushing** and **popping**.

To push a page, which will take a page and put it on top of the navigation stack (which sets it as the current page), you can do something like this:

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

This uses the reference to the **NavController** we created before, and all you need to supply to it is a reference to the page that you want to navigate to, which you will need to make sure you also import at the top of the file, like this:

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

as well as adding it to both the entryComponents and declarations arrays in **app.module.ts** and that’s it, your app should switch to the new page whenever the push code is triggered. When you push a page, a ‘Back’ button will automatically be added to the nav bar (assuming you have one), so you often don’t need to worry about using pop to navigate back to the previous page since the ‘Back’ button does this automatically for you.

There may be circumstances where you do want to manually pop a page oﬀ of the navigation stack though, in which case you can use this:

**this**.nav.pop();

Easy enough right? As I mentioned before there is still another way to change the page and that is by setting the root page. If you take a look at your **src/app/app.component.ts** file you will notice the following line:

rootPage: any = MyPage;

Declaring rootPage in the root component will set the root page, and that’s because the template for the root component looks like this:

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

So we’re setting the root property on <ion-nav> to be whatever rootPage is defined as. To change the root page at any point throughout the application, you can use our friend the **NavController** – all you have to do is call the setRoot function like this:

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

**Passing Data Between Pages**

A common requirement of mobile applications is to be able to pass data between pages. One really common example is when using the “Master Detail” pattern, which is basically where you have a list of items and then you click on one to go to another page where it displays more details about that item. When navigating to the detail page, we’re going to need to know which item we are displaying data for, which will involve passing in data from the previous page. In Ionic 2 this can be done using **NavParams**. First, you must pass through the data you want within the push call (this can also be done when using setRoot):

**this**.nav.push(SecondPage, {

thing1: data1,

thing2: data2

});

This is exactly the same as what we were doing before, except now there is an extra parameter which is an object that contains the data we want to send through to **SecondPage**. Then on the receiving page we need to import **NavParams** and inject it into our constructor:

**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){

}

}

Then you can grab the data that was passed through by doing the following:

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

**Navigation Components**

Some of the components that Ionic provides also eﬀect navigation in some way. These aren’t really core navigation concepts, but they will have an impact on navigation in your application. So let’s cover what these are and when you might want to use them.

**Modals**

You’re probably familiar with the concept of a modal already. In web development, a modal is basically some box that pops up on the screen and covers the content behind it. Usually modals have the “lightbox” style, with a blacked out background and the focus on the content area.

A Modal in Ionic is similar, in that it pops up on top of your content, but it doesn’t actually look any diﬀerent to a normal page. Generally you would want to use a modal, rather than pushing a page, when you want to give the user the ability to launch and then dismiss (close) a view, rather than navigating back to the previous page.

One cool thing about Modals are that they give you the ability to pass some data back to the page that launched it when the Modal is dismissed. For example, you can create a Modal like this (remember to import and inject **ModalController** as well!):

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

myModal.present();

Notice that the modal is “presented”, rather than being pushed onto the navigation stack. Now if we wanted to allow some data to be passed back from that modal, we could add an onDidDismiss handler to it before presenting it:

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

myModal.onDidDismiss(data => {

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

});

myModal.present();

Now when the modal is dismissed it will pass back a data object that we can do something with. To dismiss a modal, all you have to do is call the following code inside of the Modal:

**this**.view.dismiss();

where **this**.view is a reference to the **ViewController** which is kind of like the **NavController** and also needs to be imported and injected into your constructor. If we want to pass data back to that onDidDismiss handler though, we will need to do something like this:

**let** data = {

thing1: "value1",

thing2: "value2"

};

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

Now the data object will be passed back to the onDidDismiss handler from the page that launched the modal.

**Tabs**

Tabs are a very popular component that have a big impact on how navigation works in your application.

Using tabs is really simple, basically in your template you will create something like this:

<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>

and then in your class definition you just define the pages to be used as the tabs like this:

tab1Root: any = TabOne;

tab2Root: any = TabTwo;

tab3Root: any = TabThree;

**constructor**(){

}

Notice that each tab has its own **root page**. You can think of switching tabs as switching between diﬀerent root pages, and then you can push and pop pages in each tab. With a tab layout, you can switch between diﬀerent tabs, but each tab will still maintain its own history.

**Sidemenu**

A side menu doesn’t really do anything out of the ordinary in terms of navigation, the side menu is really just a UI element but it’s a convenient, and common, place to add buttons that allow the user to navigate to another page (the actual switching of pages is just done manually with **setRoot** or **push** though). Adding a side menu to your application is super easy, you just need to modify the template of your root component to include it like this:

<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>

We use <ion-menu> to create a menu, and we also have to tell it what to attach itself to. This is why we set the [content] property to content which is a reference to the local variable we created on the <ion-nav> by adding #content. So this is basically saying that the <ion-nav> is our main content area, and we want the menu to attach to that.

There is a bit to learn about navigation in Ionic 2, but once you’ve got a handle on the basics discussed in this lesson you should be able to get by in most circumstances without too much trouble.