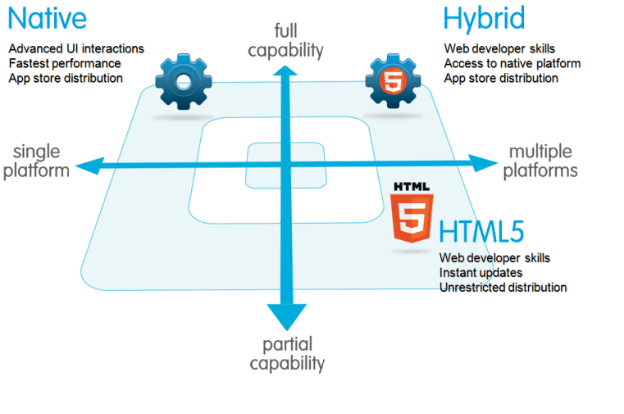
**Unit-3**

**Ionic – 1**

**Introduction to iconic platform for hybrid mobile application development**

Ionic is a toolkit, which provides us functionality to create mobile apps, using Web technologies like HTML, CSS, JavaScript and AngularJS for both Android and iOS mobiles. Ionic provides tools and Services to create a hybrid mobile Application but looks like a native Application.   
  
Ionic is a front end tool, which is built over Cordova platform. Cordova provides API for Native devices to interact with JavaScript code and Ionic framework is like a front end tool, which handles how a native app looks and feels.  
  
Before hybrid applications, if we wanted to create any Application for Android platforms, then we needed the knowledge of Java language or if we wanted to create any Application for iOS, then we needed knowledge of Objects of C or Swift programming language .   
  
If we had a need to create mobile Applications, we were required to hire a developer for an Android platform and also one developer for iOS , which was expensive and time consuming. Now, we can create a hybrid Application, which can run on multiple platforms.  
  


If we have basic knowledge of HTML5, CSS, JavaScript and AngularJS, then we can easily build mobile Applications for Android and iOS platforms without needing any knowledge of platform specific language. Hybrid Application is a boon for Web developers because now they can develop a hybrid Application, which is similar to a Web Application.   
  
We can define a hybrid Application as a Web Application, which is built using HTML5, CSS, JavaScript and after that wrapped in native container providing access to native platforms to access the hybrid app.

**Ionic** is a front-end HTML framework built on top of **AngularJS** and **Cordova**. As per their official document, the definition of this Ionic Open Source Framework is as follows –

Ionic is an **HTML5 Mobile App Development Framework** targeted at building hybrid mobile apps. Think of Ionic as the front-end UI framework that handles all the look and feel and UI interactions your app needs to be compelling. Kind of like "Bootstrap for Native", but with the support for a broad range of common native mobile components, slick animations and a beautiful design.

**Ionic Framework Features**

Following are the most important features of Ionic −

* **AngularJS** − Ionic is using AngularJS MVC architecture for building rich single page applications optimized for mobile devices.
* **CSS components** − With the native look and feel, these components offer almost all elements that a mobile application needs. The components’ default styling can be easily overridden to accommodate your own designs.
* **JavaScript components** − These components are extending CSS components with JavaScript functionalities to cover all mobile elements that cannot be done only with HTML and CSS.
* **Cordova Plugins** − Apache Cordova plugins offer API needed for using native device functions with JavaScript code.
* **Ionic CLI** − This is NodeJS utility powered with commands for starting, building, running and emulating Ionic applications.
* **Ionic View** − Very useful platform for uploading, sharing and testing your application on native devices.
* **Licence** − Ionic is released under MIT license.

**Ionic Framework Advantages**

Following are some of the most commonly known Ionic Framework Advantages −

* Ionic is used for Hybrid App Development. This means that you can package your applications for IOS, Android, Windows Phone and Firefox OS, which can save you a lot of working time.
* Starting your app is very easy since Ionic provides useful pre-generated app setup with simple layouts.
* The apps are built in a very clean and modular way, so it is very maintainable and easy to update.
* Ionic Developers Team have a very good relationship with the Google Developers Team and they are working together to improve the framework. The updates are coming out regularly and Ionic support group is always willing to help when needed.

**Ionic Framework Limitations**

Following are some of the most important Ionic Framework Limitations −

* Testing can be tricky since the browser does not always give you the right information about the phone environment. There are so many different devices as well as platforms and you usually need to cover most of them.
* It can be hard to combine different native functionalities. There will be many instances where you would run into plugin compatibility issues, which leads to build errors that are hard to debug.
* Hybrid apps tend to be slower than the native ones. However, since the mobile technologies are improving fast this will not be an issue in the future.

**Step by step installation of iconic**

The following table contains the list of components needed to start with Ionic.

|  |  |
| --- | --- |
| **Sr.No.** | **Software & Description** |
| 1 | **NodeJS**  This is the base platform needed to create Mobile Apps using Ionic. You can find detail on the NodeJS installation in our [**NodeJS Environment Setup**](https://www.tutorialspoint.com/nodejs/nodejs_environment_setup.htm). Make sure you also install npm while installing NodeJS. |
| 2 | **Android SDK**  If you are going to work on a Windows platform and are developing your apps for the Android platform, then you should have Android SDK setup on your machine. The following link has detailed information on the [**Android Environment Setup**](https://www.tutorialspoint.com/android/android_environment_setup.htm). |
| 3 | **XCode**  If you are going to work on the Mac platform and are developing your apps for the iOS platform, then you should have XCode setup on your machine. The following link has detailed information on the [**iOS Environment Setup**](https://www.tutorialspoint.com/ios/ios_environment_setup.htm). |
| 4 | **cordova and Ionic**  These are the main SDKs which is needed to start working with Ionic. This chapter explains how to setup Ionic in simple step assuming you already have the required setup as explained in the table above. |

**Installing Cordova and Ionic**

We will use the Windows command prompt for this tutorial. The same steps can be applied to the OSX terminal. Open your command window to install Cordova and Ionic −

C:\Users\Username> npm install -g cordova ionic

**Creating Apps**

While creating apps in Ionic, you can choose from the following three options to start with −

* Tabs App
* Blank App
* Side menu app

In your command window, open the folder where you want to create the app and try one of the options mentioned below.

**Tabs App**

If you want to use the Ionic tabs template, the app will be built with the tab menu, header and a couple of useful screens and functionalities. This is the default Ionic template. Open your command window and choose where you want to create your app.

C:\Users\Username> cd Desktop

This command will change the working directory. The app will be created on the Desktop.

C:\Users\Username\Desktop> ionic start myApp tabs

Ionic **Start** command will create a folder named **myApp** and setup Ionic files and folders.

C:\Users\Username\Desktop> cd myApp

Now, we want to access the **myApp** folder that we just created. This is our root folder.

Let us now add the Cordova project for the Android Platform and install the basic Cordova plugins as well. The following code allows us to run the app on the Android emulator or a device.

C:\Users\Username\Desktop\myApp> ionic platform add android

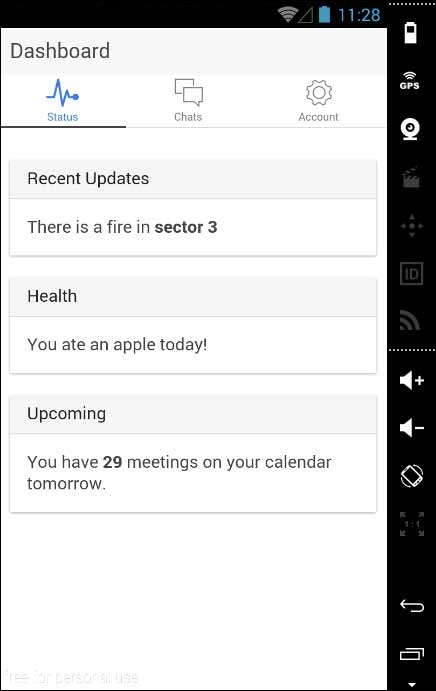
The next step is to build the app. If you have building errors after running the following command, you probably did not install the Android SDK and its dependencies.

C:\Users\Username\Desktop\myApp> ionic build android

The last step of the installation process is to run your app, which will start the mobile device, if connected, or the default emulator, if there is no device connected. Android Default Emulator is slow, so I suggest you to install [Genymotion](https://www.genymotion.com/#!/) or some other popular Android Emulator.

C:\Users\Username\Desktop\myApp> ionic run android

This will produce below result, which is an Ionic Tabs App.



**Blank App**

If you want to start from the scratch, you can install the Ionic blank template. We will use the same steps that have been explained above with the addition of **ionic start myApp blank** instead of **ionic start myApp tabs** as follows.

C:\Users\Username\Desktop> ionic start myApp blank

The Ionic **Start** command will create a folder named **myApp** and setup the Ionic files and folders.

C:\Users\Username\Desktop> cd myApp

Let us add the Cordova project for the Android Platform and install the basic Cordova plugins as explained above.

C:\Users\Username\Desktop\myApp>ionic platform add android

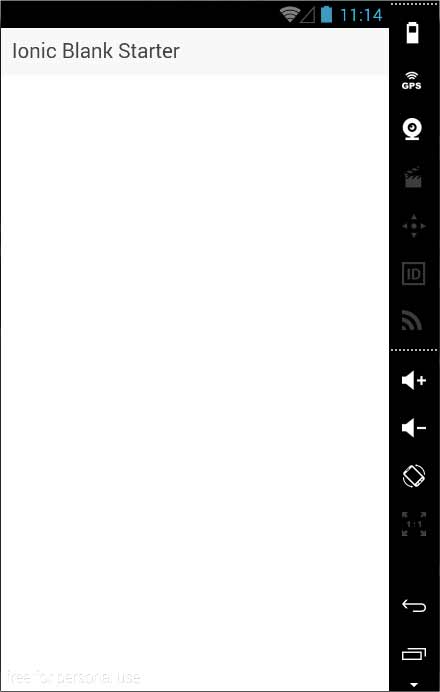
The next step is to build our app −

C:\Users\Username\Desktop\myApp> ionic build android

Finally, we will start the App as with the following code −

C:\Users\Username\Desktop\myApp> ionic run android

This will produce the following result, which is a Ionic Blank App.



**Side Menu App**

The third template that you can use is the Side Menu Template. The steps are the same as the previous two templates; we will just add **sidemenu** when starting our app as shown in the code below.

C:\Users\Username\Desktop> ionic start myApp sidemenu

The Ionic **Start** command will create a folder named **myApp** and setup the Ionic files and folders.

C:\Users\Username\Desktop> cd myApp

Let us add the Cordova project for the Android Platform and install the basic Cordova plugins with the code given below.

C:\Users\Username\Desktop\myApp> ionic platform add android

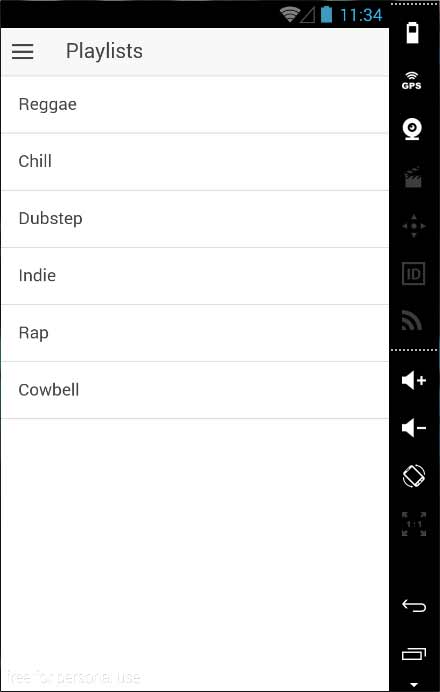
The next step is to build our app with the following code.

C:\Users\Username\Desktop\myApp> ionic build android

Finally, we will start the App with the code given below.

C:\Users\Username\Desktop\myApp> ionic run android

This will produce the following result, which is an Ionic Side Menu App.

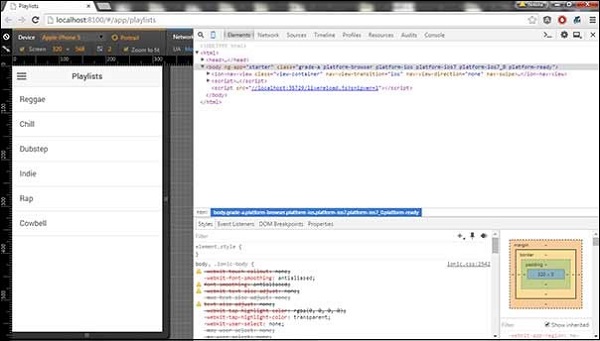


**Test App in Browser**

Since we are working with the JavaScript, you can serve your app on any web browser. This will speed up your building process, but you should always test your app on native emulators and devices. Type the following code to serve your app on the web browser.

C:\Users\Username\Desktop\myApp> ionic serve

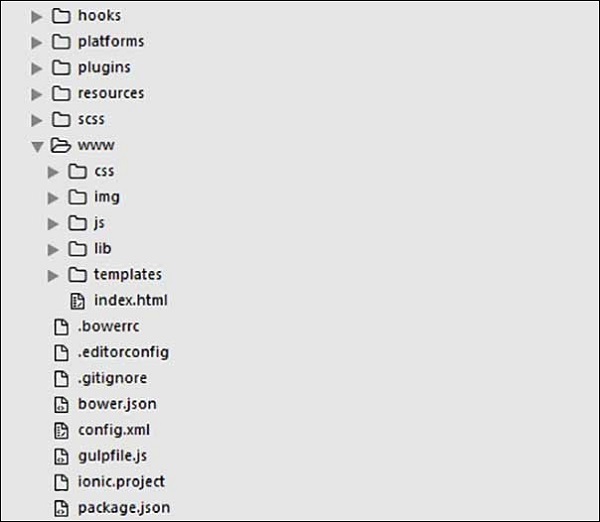
The above command will open your app in the web browser. Google Chrome provides the device mode functionality for mobile development testing. Press **F12** to access the developer console.



The top left corner of the console window click has the "Toggle Device Mode" icon. The next step will be to click "Dock to Right" icon in the top right corner. Once the page is refreshed, you should be ready for testing on the web browser.

**Project Folder Structure**

Ionic creates the following directory structure for all type of apps. This is important for any Ionic developer to understand the purpose of every directory and the files mentioned below −



Let us have a quick understanding of all the folders and files available in the project folder structure shown in the image above.

* **hooks** − Hooks are scripts that can be triggered during the build process. They are usually used for the Cordova commands customization and for building automated processes. We will not use this folder during this tutorial.
* **platforms** − This is the folder where Android and IOS projects are created. You might encounter some platform specific problems during development that will require these files, but you should leave them intact most of the time.
* **plugins** − This folder contains Cordova plugins. When you initially create an Ionic app, some of the plugins will be installed. We will show you how to install Cordova plugins in our subsequent chapters.
* **resources** − This folder is used for adding resources like icon and splash screen to your project.
* **scss** − Since Ionic core is built with Sass, this is the folder where your Sass file is located. For simplifying the process, we will not use Sass for this tutorial. Our styling will be done using CSS.
* **www** − www is the main working folder for Ionic developers. They spend most of their time here. Ionic gives us their default folder structure inside 'www', but the developers can always change it to accommodate their own needs. When this folder is opened, you will find the following sub-folders −
  + The **css** folder, where you will write your CSS styling.
  + The **img** folder for storing images.
  + The **js** folder that contains the apps main configuration file (app.js) and AngularJS components (controllers, services, directives). All your JavaScript code will be inside these folders.
  + The **libs** folder, where your libraries will be placed.
  + The **templates** folder for your HTML files.
  + **Index.html** as a starting point to your app.
* **Other Files** − Since this is a beginner’s tutorial, we will just mention some of the other important files and their purposes as well.
  + **.bowerrc** is the bower configuration file.
  + **.editorconfig** is the editor configuration file.
  + **.gitignore** is used to instruct which part of the app should be ignored when you want to push your app to the Git repository.
  + **bower.json** will contain the bower components and dependencies, if you choose to use the bower package manager.
  + **gulpfile.js** is used for creating automated tasks using the gulp task manager.
  + **config.xml** is the Cordova configuration file.
  + **package.json** contains the information about the apps, their dependencies and plugins that are installed using the NPM package manager.

**Command line interface handling of iconic-1**

The Ionic Command Line Interface (CLI) is your go-to tool for developing Ionic apps. You can follow CLI development on [Github](https://github.com/ionic-team/ionic-cli).

**Installation**

Please make sure latest [Node](https://ionicframework.com/docs/resources/what-is/#node) 6 LTS and [NPM](https://ionicframework.com/docs/resources/what-is/#npm) 3+ are installed.

Then, install the CLI globally (you may need sudo):

$ npm install -g ionic@latest

You can verify your installation with the ionic --version command.

**Creating Project in IONIC**

Start a new Ionic project using ionic start:

$ ionic start myNewProject

ionic start will prompt you to select a “starter”. We recommend using the tutorial starter for your first app. See [Starter Templates](https://ionicframework.com/docs/cli/starters.html) for a full list.

After selecting a starter, the CLI will create a new app named myNewProject. Once you cd into your project’s directory, a few new commands become available to you, such as ionic serve:

$ cd ./myNewProject

$ ionic serve

While running ionic serve, changes you make to your app code will automatically refresh the browser. If you want to see your app on a device or emulator, you can [use Cordova](https://ionicframework.com/docs/cli/#using-cordova).

You can list available commands with the ionic --help command.

**Using Cordova**

Integrate Ionic with [Cordova](https://cordova.apache.org/) to bring native capabilities to your app.

$ npm install -g cordova

$ ionic cordova --help

$ ionic cordova run ios

The ionic cordova commands (aside from ionic cordova resources) wrap the Cordova CLI. You can read about the differences in each command’s --help page. To learn more about the commands, see the [Cordova CLI Reference](https://cordova.apache.org/docs/en/latest/reference/cordova-cli/) documentation.

* For iOS development, see the [iOS Platform Guide](https://cordova.apache.org/docs/en/latest/guide/platforms/ios/index.html).
* For Android development, see the [Android Platform Guide](https://cordova.apache.org/docs/en/latest/guide/platforms/android/index.html).

**Troubleshooting**

If you’re having trouble with the Ionic CLI, you can try the following:

* Make sure you’re on the latest version of the CLI. Update with npm update -g ionic.
* Try running commands with the --verbose flag, which will print DEBUG messages.

**Component of iconic-1**

**Colors**

**Ionic Color Classes**

Ionic framework gives us a set of **nine predefined color classes**. You can use these colors or you can override it with your own styling.

The following table shows the default set of nine colors provided by Ionic. We will use these colors for styling different Ionic elements in this tutorial. For now, you can check all the colors as shown below –

|  |  |  |
| --- | --- | --- |
| **Class** | **Description** | **Result** |
| **light** | To be used for white color |  |
| **stable** | To be used for light grey color |  |
| **positive** | To be used for blue color |  |
| **calm** | To be used for light blue color |  |
| **balanced** | To be used for green color |  |
| **energized** | To be used for yellow color |  |
| **assertive** | To be used for red color |  |
| **royal** | To be used for violet color |  |
| **dark** | To be used for black color |  |

**Ionic Color Usage**

Ionic makes use of different classes for each element. For example, a header element will have **bar** class and a button will have a **button** class. To simplify the usage, we use different colors by prefixing element class in a color name.

For example, to create a blue color header, we will use a **bar-calm** as follows −

<div class = "bar bar-header bar-calm">

...

</div>

Similarly, to create a grey color button, we will use **button-stable** class as follows.

<div class = "button button-stable">

...

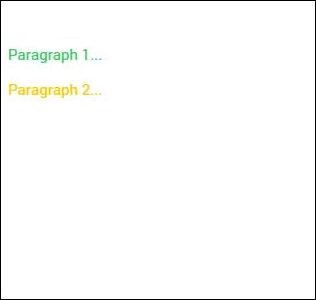
</div>

You can also use Ionic color class like any other CSS class. We will now style two paragraphs with a balanced (green) and an energized (yellow) color.

<p class = "balanced">Paragraph 1...</p>

<p class = "energized">Paragraph 2...</p>

The above code will produce the following screen −



**Customizing Colors with CSS**

When you want to change some of the Ionic default colors using CSS, you can do it by editing the **lib/css/ionic.css** file. In some cases, this approach is not very productive because every element (header, button, footer...) uses its own classes for styling.

Therefore, if you want to change the color of the "light" class to orange, you would need to search through all the elements that use this class and change it. This is useful when you want to change the color of a single element, but not very practical for changing color of all elements because it would use too much time.

**header**

The **Ionic header bar** is located on top of the screen. It can contain title, icons, buttons or some other elements on top of it. There are predefined classes of headers that you can use.

**Adding Header**

The main class for all the bars you might use in your app is **bar**. This class will always be applied to all the bars in your app. All **bar subclasses** will use the prefix – **bar**.

If you want to create a header, you need to add **bar-header** after your main **bar** class. Open your **www/index.html** file and add the header class inside your **body** tag. We are adding a header to the **index.html body** because we want it to be available on every screen in the app.

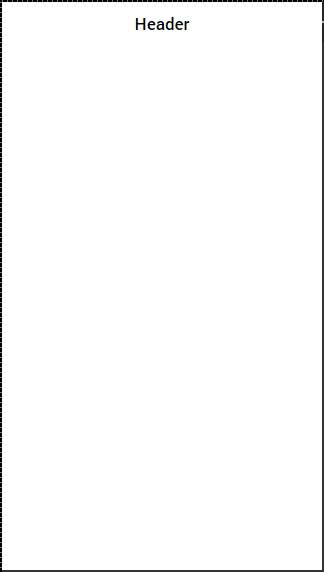
Since **bar-header** class has default (white) styling applied, we will add the title on top of it, so you can differentiate it from the rest of your screen.

<div class = "bar bar-header">

<h1 class = "title">Header</h1>

</div>

The above code will produce the following screen −



**Header Colors**

If you want to style your header, you just need to add the appropriate color class to it. When you style your elements, you need to add your main element class as prefix to your color class. Since we are styling the header bar, the prefix class will be **bar** and the color class that we want to use in this example is **positive** (blue).

<div class = "bar bar-header bar-positive">

<h1 class = "title">Header</h1>

</div>

The above code will produce the following screen −



You can use any of the following nine classes to give a color of your choice to your app header –

|  |  |  |
| --- | --- | --- |
| **Color Class** | **Description** | **Result** |
| **bar-light** | To be used for white color |  |
| **bar-stable** | To be used for light grey color |  |
| **bar-positive** | To be used for blue color |  |
| **bar-calm** | To be used for light blue color |  |
| **bar-balanced** | To be used for green color |  |
| **bar-energized** | To be used for yellow color |  |
| **bar-assertive** | To be used for red color |  |
| **bar-royal** | To be used for violet color |  |
| **bar-dark** | To be used for black color |  |

**Header Elements**

We can add other elements inside the header. The following code is an example to add a **menu** button and a **home** button inside a header. We will also add icons on top of our header buttons.

<div class = "bar bar-header bar-positive">

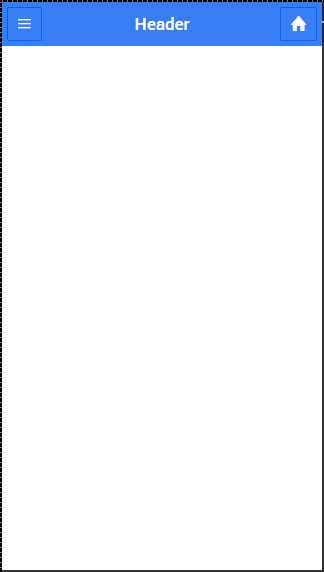
<button class = "button icon ion-navicon"></button>

<h1 class = "title">Header Buttons</h1>

<button class = "button icon ion-home"></button>

</div>

The above code will produce the following screen −



**Sub Header**

You can create a sub header that will be located just below the header bar. The following example will show how to add a header and a sub header to your app. Here, we have styled the sub header with an "assertive" (red) color class.

<div class = "bar bar-header bar-positive">

<button class = "button icon ion-navicon"></button>

<h1 class = "title">Header Buttons</h1>

<button class = "button icon ion-home"></button>

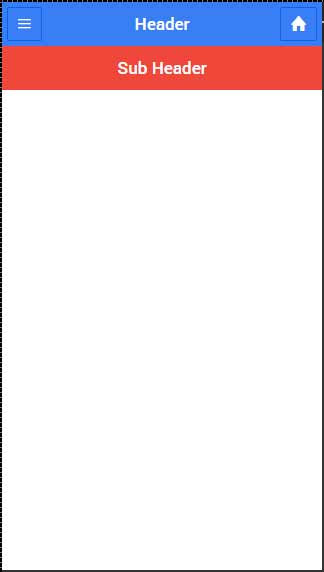
</div>

<div class = "bar bar-subheader bar-assertive">

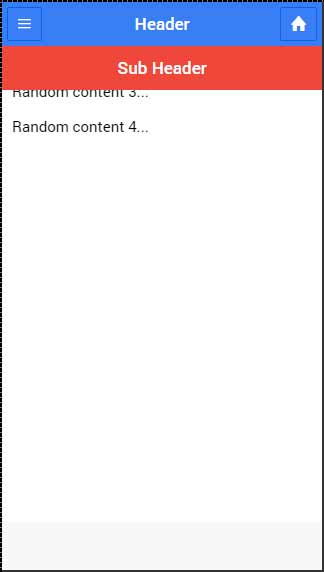
<h2 class = "title">Sub Header</h2>

</div>

The above code will produce the following screen −



When your route is changed to any of the app screens, you will notice that the header and the sub header are covering some content as shown in the screenshot below.



To fix this you need to add a **‘has-header’** or a **‘has-subheader’** class to the **ion-content** tags of your screens. Open one of your HTML files from **www/templates** and add the **has-subheader** class to the **ion-content**. If you only use header in your app, you will need to add the **has-header** class instead.

<ion-content class = "padding has-subheader">

The above code will produce the following screen −



**Button**

There are several types of buttons in the Ionic Framework and these buttons are subtly animated, which further enhances the user experience when using the app. The main class for all the button types is **button**. This class will always be applied to our buttons, and we will use it as a prefix when working with sub classes.

**Block Buttons**

Block buttons will always have 100% width of their parent container. They will also have a small padding applied. You will use **button-block** class for adding block buttons. If you want to remove padding but keep the full width, you can use the **button-full** class.

Following is an example to show the usage of both classes −

<button class = "button button-block">

button-block

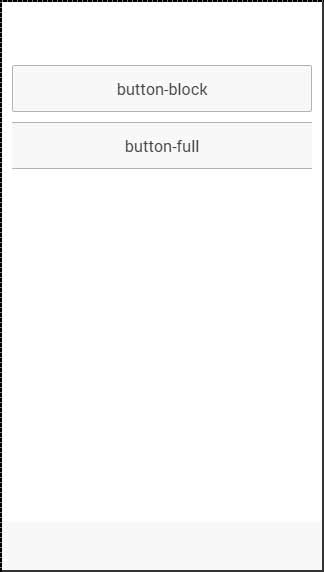
</button>

<button class = "button button-full">

button-full

</button>

The above code will produce the following screen −



**Button Size**

There are two Ionic classes for changing the button size −

* **button-small** and
* **button-large**.

Following is an example to show their usage −

<button class = "button button-small">

button-small

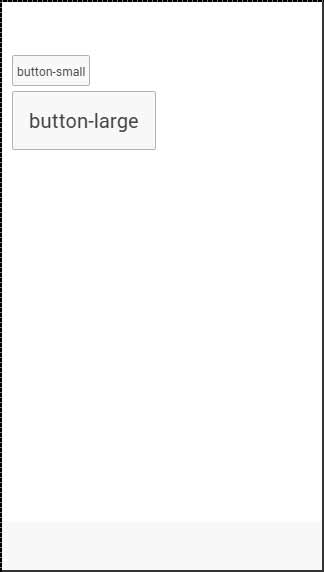
</button>

<button class = "button button-large">

button-large

</button>

The above code will produce the following screen −



**Button Colors**

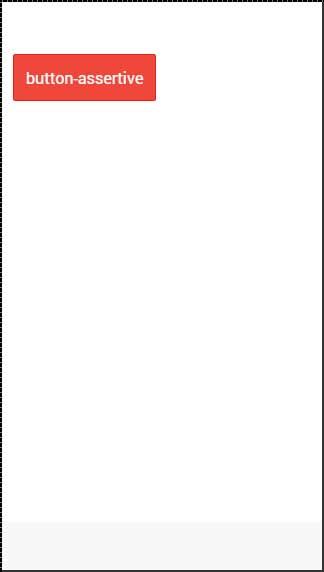
If you want to style your button, you just need to add appropriate color class to it. When you style your elements, you need to add your main element class as a prefix to your color class. Since we are styling the footer bar, the prefix class will be a **bar** and the color class that we want to use in this example is **assertive** (red).

<button class = "button button-assertive">

button-assertive

</button>

The above code will produce the following screen −



You can use any of the following nine classes to give a color of your choice to your app buttons −

|  |  |  |
| --- | --- | --- |
| **Color Class** | **Description** | **Result** |
| **button-light** | To be used for white color |  |
| **button-stable** | To be used for light grey color |  |
| **button-positive** | To be used for blue color |  |
| **button-calm** | To be used for light blue color |  |
| **button-balanced** | To be used for green color |  |
| **button-energized** | To be used for yellow color |  |
| **button-assertive** | To be used for red color |  |
| **button-royal** | To be used for violet color |  |
| **button-dark** | To be used for black color |  |

**Button Outline**

If you want your buttons transparent, you can apply **button-outline** class. Buttons with this class will have an outline border and a transparent background.

To remove the border from the button, you can use the **button-clear** class. The following example shows how to use these two classes.

<button class = "button button-assertive button-outline">

button-outline

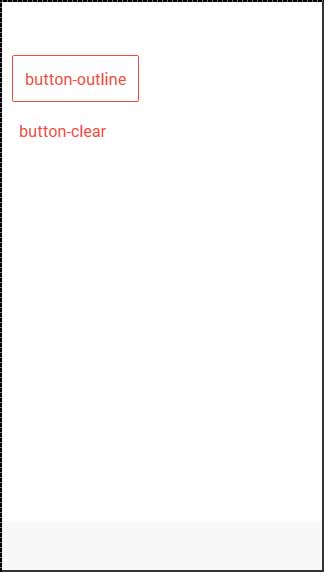
</button>

<button class = "button button-assertive button-clear">

button-clear

</button>

The above code will produce the following screen −



**Adding Icons**

When you want to add Icons to your buttons, the best way is to use the **icon**class. You can place the icon on one side of the button by using the **icon-left**or the **icon-right**. You will usually want to move your icon to one side when you have some text on top of your button as explained below.

<button class = "button icon ion-home">

</button>

<button class = "button icon icon-left ion-home">

Home

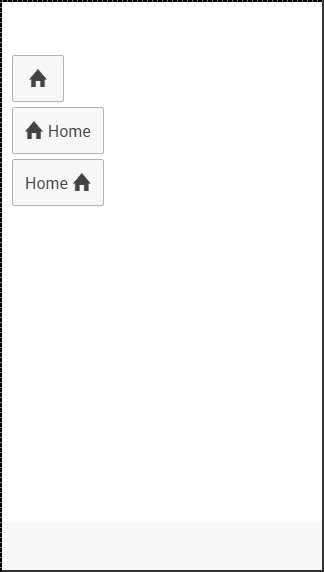
</button>

<button class = "button icon icon-right ion-home">

Home

</button>

The above code will produce the following screen −



**Button Bar**

If you want to group a couple of buttons together, you can use the **button-bar** class. The buttons will have equal size by default.

<div class = "button-bar">

<a class = "button button-positive">1</a>

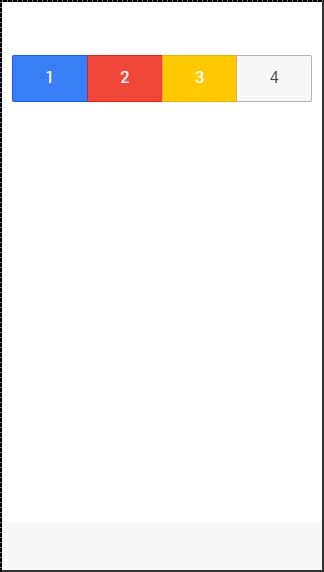
<a class = "button button-assertive">2</a>

<a class = "button button-energized">3</a>

<a class = "button">4</a>

</div>

The above code will produce the following screen −



**List**

**Lists** are one of the most popular elements of any web or mobile application. They are usually used for displaying various information. They can be combined with other HTML elements to create different menus, tabs or to break the monotony of pure text files. Ionic framework offers different list types to make their usage easy.

**Creating Ionic List**

Every list is created with two elements. When you want to create a basic list your **<ul>** tag needs to have the **list** class assigned, and your **<li>** tag will use the **item** class. Another interesting thing is that you do not even need to use **<ul>**, **<ol>** and **<li>** tags for your lists. You can use any other elements, but the important thing is to add **list** and **item** classes appropriately.

<div class = "list">

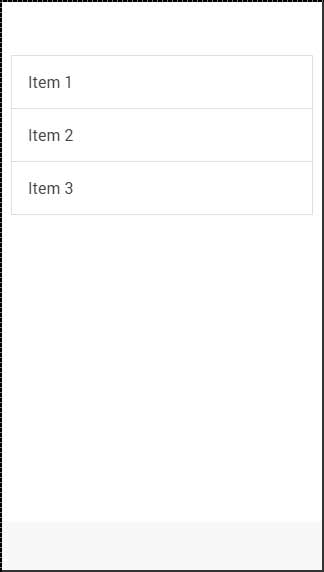
<div class = "item">Item 1</div>

<div class = "item">Item 2</div>

<div class = "item">Item 3</div>

</div>

The above code will produce the following screen −



**Inset List**

When you need a list to fill your own container, you can add the **list-insets**after your **list** class. This will add some margin to it and it will adjust the list size to your container.

<div class = "list list-inset">

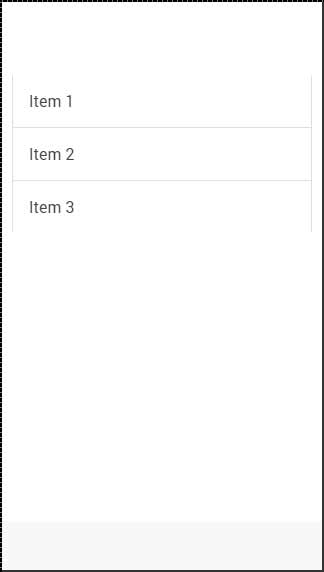
<div class = "item">Item 1</div>

<div class = "item">Item 2</div>

<div class = "item">Item 3</div>

</div>

The above code will produce the following screen −



**Item Dividers**

Dividers are used for organizing some elements into logical groups. Ionic gives us **item-divider** class for this. Again, like with all the other Ionic elements, we just need to add the **item-divider** class after the **item** class. Item dividers are useful as a list header, since they have stronger styling than the other list items by default.

<div class = "list">

<div class = "item item-divider">Item Divider 1</div>

<div class = "item">Item 1</div>

<div class = "item">Item 2</div>

<div class = "item">Item 3</div>

<div class = "item item-divider">Item Divider 2</div>

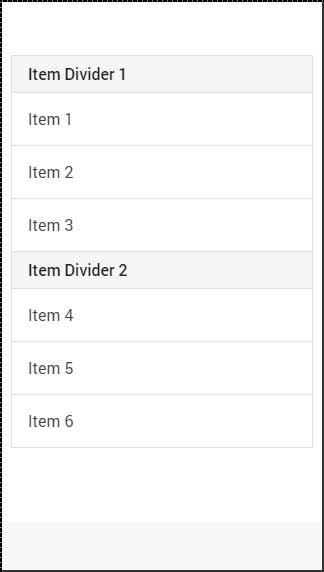
<div class = "item">Item 4</div>

<div class = "item">Item 5</div>

<div class = "item">Item 6</div>

</div>

The above code will produce the following screen −



**Adding Icons**

We already showed you how to add icons to your buttons. When adding icons to the list items, you need to choose what side you want to place them. There are **item-icon-left** and **item-icon-right** classes for this. You can also combine those two classes, if you want to have your Icons on both the sides. Finally, there is the **item-note** class to add a text note to your item.

<div class = "list">

<div class = "item item-icon-left">

<i class = "icon ion-home"></i>

Left side Icon

</div>

<div class = "item item-icon-right">

<i class = "icon ion-star"></i>

Right side Icon

</div>

<div class = "item item-icon-left item-icon-right">

<i class = "icon ion-home"></i>

<i class = "icon ion-star"></i>

Both sides Icons

</div>

<div class = "item item-icon-left">

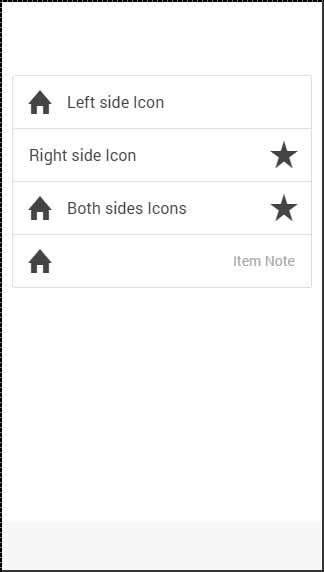
<i class = "icon ion-home"></i>

<span class = "text-note">Text note</span>

</div>

</div>

The above code will produce the following screen −



**Adding Avatars and Thumbnails**

Avatars and thumbnails are similar. The main difference is that avatars are smaller than thumbnails. These thumbnails are covering most of the full height of the list item, while avatars are medium sized circle images. The classes that are used are **item-avatar** and **item-thumbnail**. You can also choose which side you want to place your avatars and thumbnails as shown in the thumbnail code example below.

<div class = "list">

<div class = "item item-avatar">

<img src = "my-image.png">

<h3>Avatar</h3>

</div>

<div class = "item item-thumbnail-left">

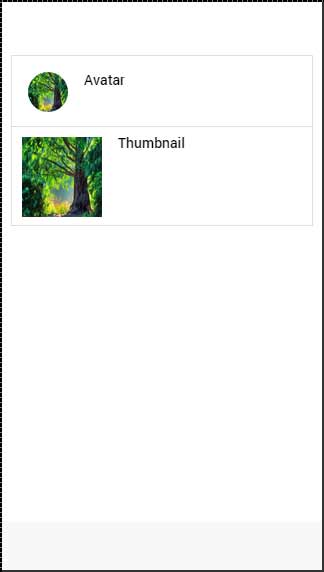
<img src = "my-image.png">

<h3>Thumbnail</h3>

</div>

</div>

The above code will produce the following screen −



**Card**

Since mobile devices have smaller screen size, cards are one of the best elements for displaying information that will feel user friendly. In the previous chapter, we have discussed how to inset lists. Cards are very similar to inset lists, but they offer some additional shadowing that can influence the performance for larger lists.

**Adding Cards**

A default card can be created by adding a **card** class to your element. Cards are usually formed as lists with the **item** class. One of the most useful class is the **item-text-wrap**. This will help when you have too much text, so you want to wrap it inside your card. The first card in the following example does not have the **item-text-wrap** class assigned, but the second one is using it.

<div class = "card">

<div class = "item">

This is a Ionic card without item-text-wrap class.

</div>

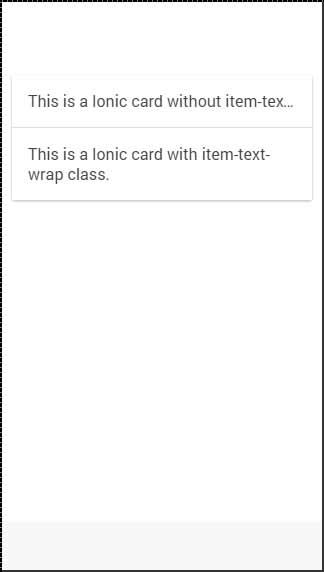
<div class = "item item-text-wrap">

This is a Ionic card with item-text-wrap class.

</div>

</div>

The above code will produce the following screen −



**Card Header and Footer**

In the previous chapter, we have already discussed how to use the **item-divider** class for grouping lists. This class can be very useful when working with cards to create card headers. The same class can be used for footers as shown in the following code –

<div class = "card list">

<div class = "item item-divider">

Card header

</div>

<div class = "item item-text-wrap">

Card text...

</div>

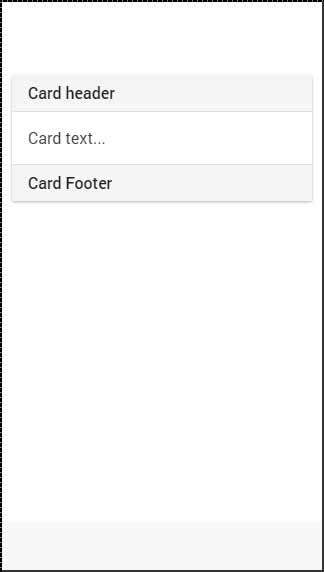
<div class = "item item-divider">

Card Footer

</div>

</div>

The above code will produce the following screen −



**Complete Card**

You can add any element on top of your card. In following example, we will show you how to use the **full-image** class together with the **item-body** to get a good-looking windowed image inside your card.

<div class = "card">

<div class = "item item-avatar">

<img src = "my-image.png">

<h2>Card Name</h2>

</div>

<div class = "item item-body">

<img class = "full-image" src = "my-image.png">

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque eget

pharetra tortor. Proin quis eros imperdiet, facilisis nisi in, tincidunt orci.

Nam tristique elit massa, quis faucibus augue finibus ac.

</div>

</div>

The above code will produce the following screen −



**Forms**

**Ionic forms** are mostly used for interaction with users and collecting needed info. This chapter will cover various text input forms and in our subsequent chapters, we will explain how to use other form elements using the Ionic framework.

**Using Input Form**

The best way to use forms is to use **list** and **item** as your main classes. Your app will usually consist more than one-form element, so it makes sense to organize it as a list. In the following example, you can notice that the item element is a **label** tag.

You can use any other element, but a label will provide the ability to tap on any part of the element to focus your text input. You can set a **placeholder**that will look different from the input text and it will be hidden as soon as you start typing. You can see this in the example below.

<div class = "list">

<label class = "item item-input">

<input type = "text" placeholder = "Placeholder 1" />

</label>

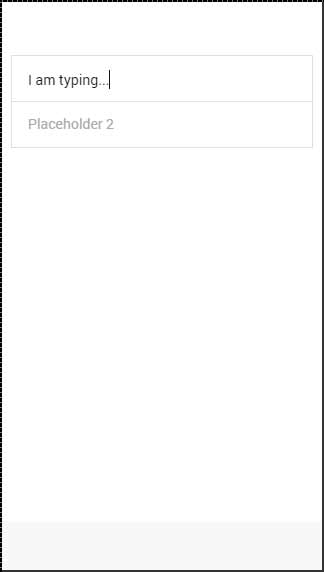
<label class = "item item-input">

<input type = "text" placeholder = "Placeholder 2" />

</label>

</div>

The above code will produce the following screen −



**Ionic Labels**

Ionic offers some other options for your label. You can use the **input-label**class, if you want your placeholder to be on the left side when you type the text.

<div class = "list">

<label class = "item item-input">

<input type = "text" placeholder = "Placeholder 1" />

</label>

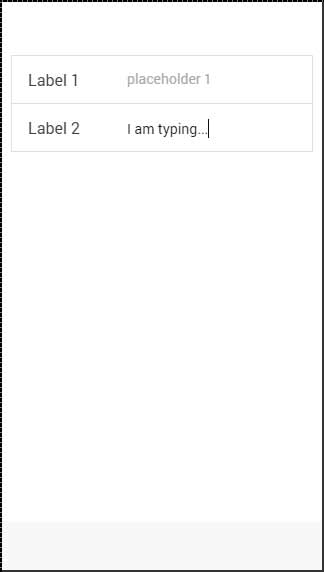
<label class = "item item-input">

<input type = "text" placeholder = "Placeholder 2" />

</label>

</div>

The above code will produce the following screen −



Stacked Label

**Stacked label** is the other option that allows moving your label on top or the bottom of the input. To achieve this, we will add the **item-stacked-label**class to our label element and we need to create a new element and assign the **input-label** class to it. If we want it to be on top, we just need to add this element before the **input** tag. This is shown in the following example.

Notice that the **span** tag is before the **input** tag. If we changed their places, it would appear below it on the screen.

<div class = "list">

<label class = "item item-input item-stacked-label">

<span class = "input-label">Label 1</span>

<input type = "text" placeholder = "Placeholder 1" />

</label>

<label class = "item item-input item-stacked-label">

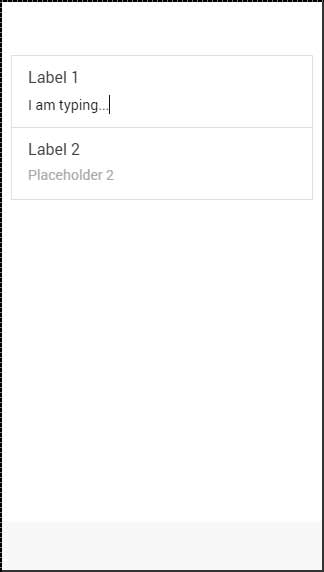
<span class = "input-label">Label 2</span>

<input type = "text" placeholder = "Placeholder 2" />

</label>

</div>

The above code will produce the following screen −



**Floating Label**

**Floating labels** are the third option we can use. These labels will be hidden before we start typing. As soon the typing starts, they will appear on top of the element with nice floating animation. We can use floating labels the same way as we used stacked labels. The only difference is that this time we will use **item-floating-label** class.

<div class = "list">

<label class = "item item-input item-floating-label">

<span class = "input-label"t>Label 1</span>

<input type = "text" placeholder = "Placeholder 1" />

</label>

<label class = "item item-input item-floating-label">

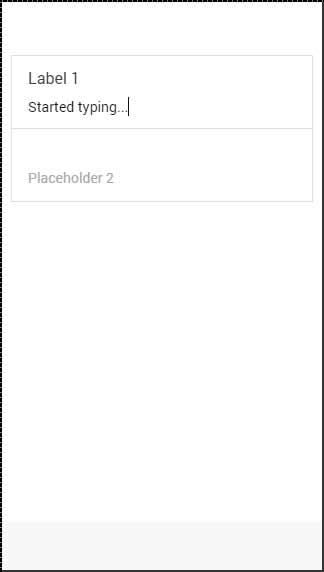
<span class = "input-label">Label 2</span>

<input type = "text" placeholder = "Placeholder 2" />

</label>

</div>

The above code will produce the following screen −



**Inset inputs**

In our last chapter, we discussed how to inset Ionic elements. You can also inset an input by adding the **item-input-inset** class to your item and the **item-input-wrapper** to your label. A Wrapper will add additional styling to your label.

If you add some other elements next to your label, the label size will adjust to accommodate the new element. You can also add elements inside your label (usually icons).

The following example shows two inset inputs. The first one has a button next to the label, and the second one has an icon inside it. We used the **placeholder-icon** class to make the icon with the same color as the placeholder text. Without it, the icon would use the color of the label.

<div class = "list">

<div class = "item item-input-inset">

<label class = "item item-input-wrapper">

<input type = "text" placeholder = "Placeholder 1" />

</label>

<button class = "button">button</button>

</div>

<div class = "item item-input-inset">

<label class = "item item-input-wrapper">

<i class = "icon ion-edit placeholder-icon"></i>

<input type = "text" placeholder = "Placeholder 2" />

</label>

</div>

</div>

The above code will produce the following screen −



**Checkgbox**

**Ionic checkbox** is almost the same as toggle. These two are styled differently but are used for the same purposes.

**Adding Checkbox**

When creating a checkbox form, you need to add the **checkbox** class name to both label and the input elements. The following example shows two simple checkboxes, one is checked and the other is not.

<label class = "checkbox">

<input type = "checkbox">

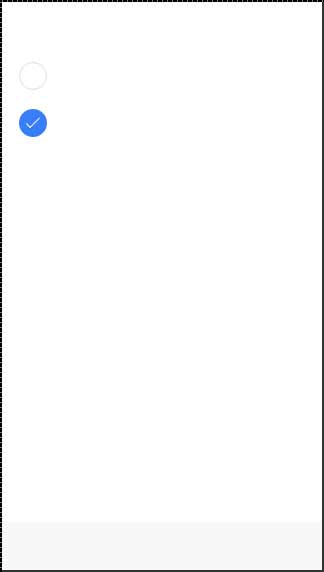
</label>

<label class = "checkbox">

<input type = "checkbox">

</label>

The above code will produce the following screen −



**Multiple Checkboxes**

As we already showed, the list will be used for multiple elements. Now we will use the **item-checkbox** class for each list item.

<ul class = "list">

<li class = "item item-checkbox">

Checbox 1

<label class = "checkbox">

<input type = "checkbox" />

</label>

</li>

<li class = "item item-checkbox">

Checkbox 2

<label class = "checkbox">

<input type = "checkbox" />

</label>

</li>

<li class = "item item-checkbox">

Checkbox e

<label class = "checkbox">

<input type = "checkbox" />

</label>

</li>

<li class = "item item-checkbox">

Checkbox 4

<label class = "checkbox">

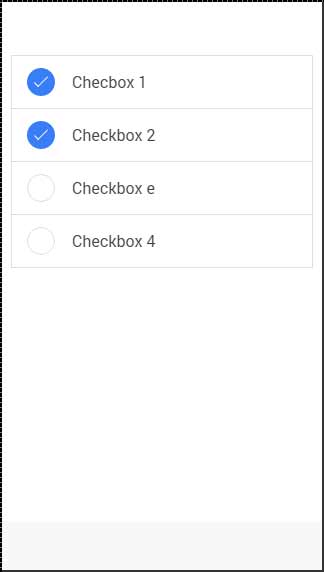
<input type = "checkbox" />

</label>

</li>

</ul>

The above code will produce the following screen −



**Styling Checkbox**

When you want to style a checkbox, you need to apply any Ionic color class with the **checkbox** prefix. Check the following example to see how it looks like. We will use the list of checkboxes for this example.

<ul class = "list">

<li class = "item item-checkbox checkbox-light">

Checbox 1

<label class = "checkbox">

<input type = "checkbox" />

</label>

</li>

<li class = "item item-checkbox checkbox-stable">

Checkbox 2

<label class = "checkbox">

<input type = "checkbox" />

</label>

</li>

<li class = "item item-checkbox checkbox-positive">

Checkbox 3

<label class = "checkbox">

<input type = "checkbox" />

</label>

</li>

<li class = "item item-checkbox checkbox-calm">

Checkbox 4

<label class = "checkbox">

<input type = "checkbox" />

</label>

</li>

<li class = "item item-checkbox checkbox-balanced">

Checkbox 5

<label class = "checkbox">

<input type = "checkbox" />

</label>

</li>

<li class = "item item-checkbox checkbox-energized">

Checkbox 6

<label class = "checkbox">

<input type = "checkbox" />

</label>

</li>

<li class = "item item-checkbox checkbox-assertive">

Checkbox 7

<label class = "checkbox">

<input type = "checkbox" />

</label>

</li>

<li class = "item item-checkbox checkbox-royal">

Checkbox 8

<label class = "checkbox">

<input type = "checkbox" />

</label>

</li>

<li class = "item item-checkbox checkbox-dark">

Checkbox 9

<label class = "checkbox">

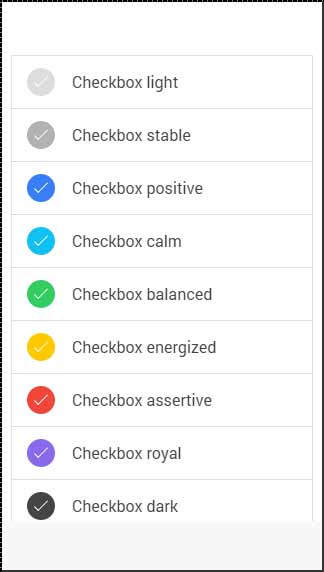
<input type = "checkbox" />

</label>

</li>

</ul>

The above code will produce the following screen −



**Radio buttons**

**Radio buttons** are another form of an element, which we will be covering in this chapter. The difference between radio buttons from toggle and checkbox forms is that when using the former, you choose only one radio button from the list. As the latter allows you to choose more than one.

**Adding Radio Buttons**

Since there will always be more than one radio button to choose from, the best way is to create a list. We did this whenever we wanted multiple elements. The list item class will be **item-radio**. Again, we will use **label** for this as we have used with all the other forms. Input will have the **name** attribute. This attribute will group all the buttons that you want to use as a possible choice. The **item-content** class is used to display options clearly. At the end, we will use the **radio-icon** class to add the checkmark icon that will be used to mark the option that the user chooses.

In the following example, there are four radio buttons and the second one is chosen.

<div class = "list">

<label class = "item item-radio">

<input type = "radio" name = "group1" />

<div class = "item-content">

Choice 1

</div>

<i class = "radio-icon ion-checkmark"></i>

</label>

<label class = "item item-radio">

<input type = "radio" name = "group1" />

<div class = "item-content">

Choice 2

</div>

<i class = "radio-icon ion-checkmark"></i>

</label>

<label class = "item item-radio">

<input type = "radio" name = "group1" />

<div class = "item-content">

Choice 3

</div>

<i class = "radio-icon ion-checkmark"></i>

</label>

<label class = "item item-radio">

<input type = "radio" name = "group1" />

<div class = "item-content">

Choice 4

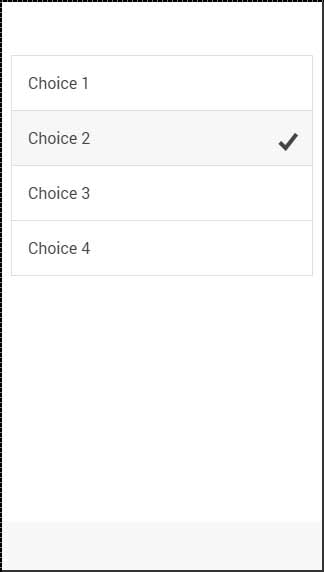
</div>

<i class = "radio-icon ion-checkmark"></i>

</label>

</div>

The above code will produce the following screen −



**Multiple Radio Button Groups**

Sometimes you want to create more than one group. This is what the **name**attribute is made for; the following example will group the first two and the last two buttons as two option groups.

We will use the **item-divider** class to separate two groups. Notice that the first group has the **name** attribute equal to **group1** and the second one uses **group2**.

<div class = "list">

<div class = " item item-divider">

Group1

</div>

<label class = "item item-radio">

<input type = "radio" name = "group1" />

<div class = "item-content">

Choice 1

</div>

<i class = "radio-icon ion-checkmark"></i>

</label>

<label class = "item item-radio">

<input type = "radio" name = "group1" />

<div class = "item-content">

Choice 2

</div>

<i class = "radio-icon ion-checkmark"></i>

</label>

<div class = " item item-divider">

Group2

</div>

<label class = "item item-radio">

<input type = "radio" name = "group2" />

<div class = "item-content">

Choice 3

</div>

<i class = "radio-icon ion-checkmark"></i>

</label>

<label class = "item item-radio">

<input type = "radio" name = "group2" />

<div class = "item-content">

Choice 4

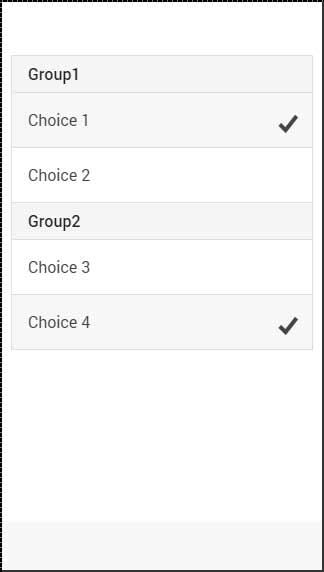
</div>

<i class = "radio-icon ion-checkmark"></i>

</label>

</div>

The above code will produce the following screen −



**Range**

Ionic range is used to choose and display the level of something. It will represent the actual value in co-relation to maximal and minimal value. Ionic offers a simple way of working with Range.

**Using Range**

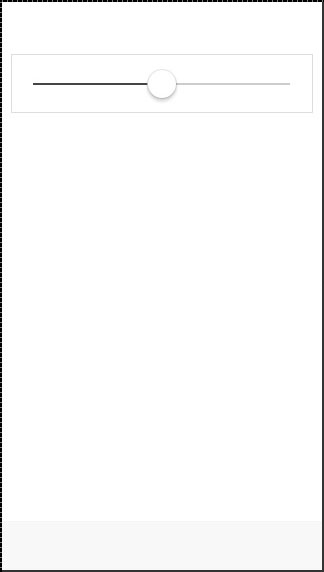
Range is used as an inside item element. The class that is used is **range**. We will place this class after the **item** class. This will prepare a container where the range will be placed. After creating a container, we need to add **input** and assign the **range** type to it and the **name** attribute as well.

<div class = "item range">

<input type = "range" name = "range1">

</div>

The above code will produce the following screen −



**Adding Icons**

Range will usually require icons to display the data clearly. We just need to add icons before and after the range input to place them on both sides of the range element.

<div class = "item range">

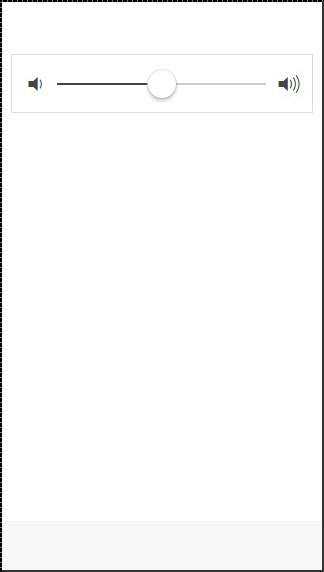
<i class = "icon ion-volume-low"></i>

<input type = "range" name = "volume">

<i class = "icon ion-volume-high"></i>

</div>

The above code will produce the following screen −



**Styling Range**

Our next example will show you how to style Range with Ionic colors. The color classes will use a **range** prefix. We will create a list with nine ranges and style it differently.

<div class = "list">

<div class = "item range range-light">

<input type = "range" name = "volume">

</div>

<div class = "item range range-stable">

<input type = "range" name = "volume">

</div>

<div class = "item range range-positive">

<input type = "range" name = "volume">

</div>

<div class = "item range range-calm">

<input type = "range" name = "volume">

</div>

<div class = "item range range-balanced">

<input type = "range" name = "volume">

</div>

<div class = "item range range-energized">

<input type = "range" name = "volume">

</div>

<div class = "item range range-assertive">

<input type = "range" name = "volume">

</div>

<div class = "item range range-royal">

<input type = "range" name = "volume">

</div>

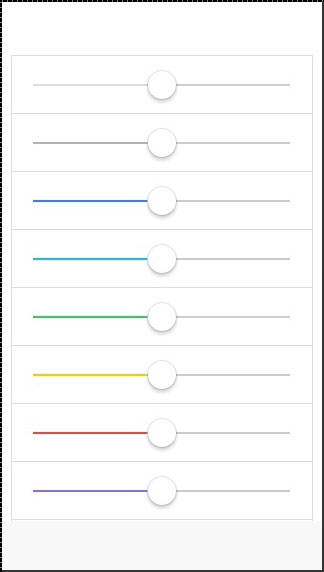
<div class = "item range range-dark">

<input type = "range" name = "volume">

</div>

</div>

The above code will produce the following screen −



**Select (dropdown)**

**Ionic Select** will create a simple menu with select options for the user to choose. This Select Menu will look differently on different platforms, since its styling is handled by the browser.

**Using Select**

First, we will create a **label** and add the **item-input** and the **item-select**classes. The second class will add additional styling to the select form and then we will add the **input-label** class inside that will be used to add a name to our select element. We will also add **select** with **option** inside. This is regular HTML5 select element. The following example is showing Ionic Select with three options.

<label class = "item item-input item-select">

<div class = "input-label">

Select

</div>

<select>

<option>Option 1</option>

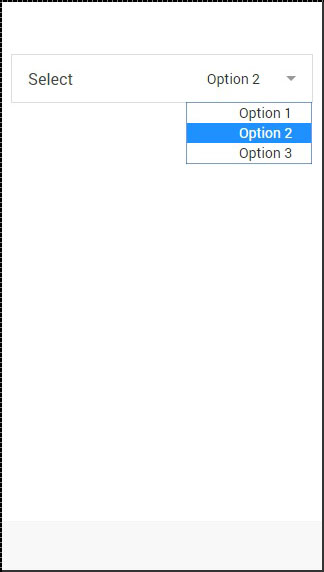
<option selected>Option 2</option>

<option>Option 3</option>

</select>

</label>

The above code will produce the following screen −



**Styling Select**

The following example will show you how to apply styling to select. We are creating a list with nine differently styled select elements using Ionic colors. Since we are using list with items, **item** will be the prefix to the color classes.

<div class = "list">

<label class = "item item-input item-select item-light">

<div class = "input-label">

Select

</div>

<select>

<option>Option 1</option>

<option selected>Option 2</option>

<option>Option 3</option>

</select>

</label>

<label class = "item item-input item-select item-stable">

<div class = "input-label">

Select

</div>

<select>

<option>Option 1</option>

<option selected>Option 2</option>

<option>Option 3</option>

</select>

</label>

<label class = "item item-input item-select item-positive">

<div class = "input-label">

Select

</div>

<select>

<option>Option 1</option>

<option selected>Option 2</option>

<option>Option 3</option>

</select>

</label>

<label class = "item item-input item-select item-calm">

<div class = "input-label">

Select

</div>

<select>

<option>Option 1</option>

<option selected>Option 2</option>

<option>Option 3</option>

</select>

</label>

<label class = "item item-input item-select item-balanced">

<div class = "input-label">

Select

</div>

<select>

<option>Option 1</option>

<option selected>Option 2</option>

<option>Option 3</option>

</select>

</label>

<label class = "item item-input item-select item-energized">

<div class = "input-label">

Select

</div>

<select>

<option>Option 1</option>

<option selected>Option 2</option>

<option>Option 3</option>

</select>

</label>

<label class = "item item-input item-select item-assertive">

<div class = "input-label">

Select

</div>

<select>

<option>Option 1</option>

<option selected>Option 2</option>

<option>Option 3</option>

</select>

</label>

<label class = "item item-input item-select item-royal">

<div class = "input-label">

Select

</div>

<select>

<option>Option 1</option>

<option selected>Option 2</option>

<option>Option 3</option>

</select>

</label>

<label class = "item item-input item-select item-dark">

<div class = "input-label">

Select

</div>

<select>

<option>Option 1</option>

<option selected>Option 2</option>

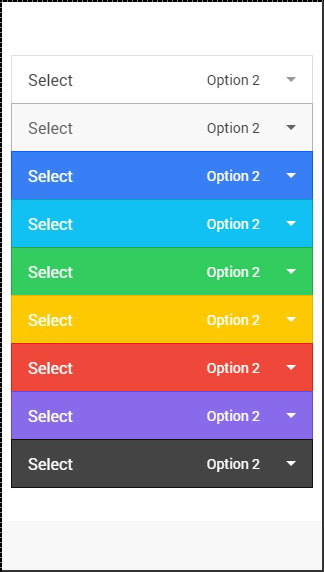
<option>Option 3</option>

</select>

</label>

</div>

The above code will produce the following screen −



**Tabs**

**Ionic tabs** are most of the time used for mobile navigation. Styling is optimized for different platforms. This means that on android devices, tabs will be placed at the top of the screen, while on IOS it will be at the bottom. There are different ways of creating tabs.

**Simple Tabs**

Simple Tabs menu can be created with a **tabs** class. For the inside element that is using this class, we need to add **tab-item** elements. Since tabs are usually used for navigation, we will use **<a>** tags for tab items. The following example is showing a menu with four tabs.

<div class = "tabs">

<a class = "tab-item">

Tab 1

</a>

<a class = "tab-item">

Tab 2

</a>

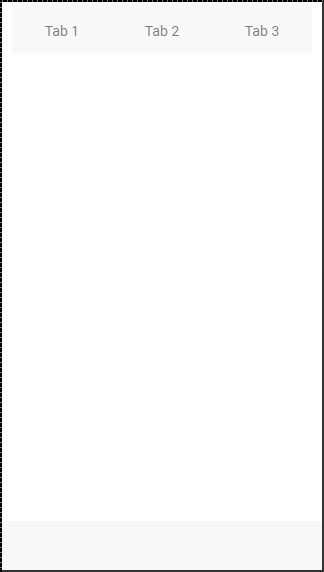
<a class = "tab-item">

Tab 3

</a>

</div>

The above code will produce the following screen −



**Adding Icons**

Ionic provides classes for adding icons to tabs. If you want your tabs to have icons without any text, a **tabs-icon-only** class should be added after the **tabs**class. Of course, you need to add icons you want to display.

<div class = "tabs tabs-icon-only">

<a class = "tab-item">

<i class = "icon ion-home"></i>

</a>

<a class = "tab-item">

<i class = "icon ion-star"></i>

</a>

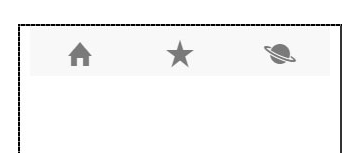
<a class = "tab-item">

<i class = "icon ion-planet"></i>

</a>

</div>

The above code will produce the following screen −



You can also add icons and text together. The **tabs-icon-top** and **tabs-icon-left** are classes that will place the icon above or on the left side respectively. Implementation is the same as the example given above, we will just add a new class and text that we want to use. The following example shows icons placed above the text.

<div class = "tabs tabs-icon-top">

<a class = "tab-item">

<i class = "icon ion-home"></i>

Tab 1

</a>

<a class = "tab-item">

<i class = "icon ion-star"></i>

Tab 2

</a>

<a class = "tab-item">

<i class = "icon ion-planet"></i>

Tab 3

</a>

</div>

The above code will produce the following screen −



**Striped Tabs**

Striped Tabs can be created by adding a container around our tabs with the **tabs-striped** class. This class allows the usage of the **tabs-background** and the **tabs-color** prefixes for adding some of the Ionic colors to the tabs menu.

In the following example, we will use the **tabs-background-positive** (blue) class to style the background of our menu, and the **tabs-color-light** (white) class to style the tab icons. Notice the difference between the second tab that is active and the other two that are not.

<div class = "tabs-striped tabs-background-positive tabs-color-light">

<div class = "tabs">

<a class = "tab-item">

<i class = "icon ion-home"></i>

</a>

<a class = "tab-item active">

<i class = "icon ion-star"></i>

</a>

<a class = "tab-item">

<i class = "icon ion-planet"></i>

</a>

</div>

</div>

The above code will produce the following screen −



**Grid**

Working with the **Ionic Grid System** is straightforward. There are two main classes – **row** for working with rows and **col** for columns.

You can choose as many columns or rows you want. All of them will adjust its size to accommodate the available space, although you can change this behavior to suit your needs.

## Simple Grid

The following example shows how to use the **col** and the **row** classes. We will create two rows. The first row will have five columns and the second one will have only three. Notice how the width of the columns is different in the first and second row.

<div class = "row">

<div class = "col">col 1</div>

<div class = "col">col 2</div>

<div class = "col">col 3</div>

<div class = "col">col 4</div>

<div class = "col">col 5</div>

</div>

<div class = "row">

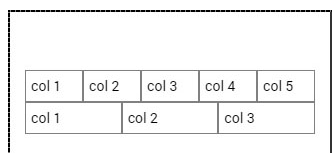
<div class = "col">col 1</div>

<div class = "col">col 2</div>

<div class = "col">col 3</div>

</div>

The above code will produce the following screen −



## Column Sizes

Sometimes you do not want to leave the column sizes automatically assigned. If this is the case, you can choose the **col** prefix followed by a number that will represent a percentage of the **row** width. This will apply only to the column with a specific size applied. The other columns will adjust to the available space that is left.

In the following example, the first column will use 50 percent of the full width and the others will adjust accordingly.

<div class = "row">

<div class = "col col-50">col 1</div>

<div class = "col">col 2</div>

<div class = "col">col 3</div>

<div class = "col">col 4</div>

<div class = "col">col 5</div>

</div>

<div class = "row">

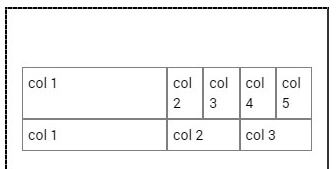
<div class = "col col-50">col 1</div>

<div class = "col">col 2</div>

<div class = "col">col 3</div>

</div>

The above code will produce following screen −



The following table shows the available percentage options that Ionic grid system provides −

### Column Percentage Classnames

|  |  |
| --- | --- |
| **Class Name** | **Percentage Used** |
| **col-10** | 10% |
| **col-20** | 20% |
| **col-25** | 25% |
| **col-33** | 33.3333% |
| **col-50** | 50% |
| **col-67** | 66.6666% |
| **col-75** | 75% |
| **col-80** | 80% |
| **col-90** | 90% |

## Horizontal and Vertical Positioning

The columns can be offset from the left. It works the same for the specific size of the columns. This time the prefix will be **col-offset** and then we will use the same percentage numbers showed in the table above. The following example shows how can we offset the second column of both the rows by 25 percent.

<div class = "row">

<div class = "col">col 1</div>

<div class = "col col-offset-25">col 2</div>

<div class = "col">col 3</div>

<div class = "col">col 4</div>

<div class = "col">col 5</div>

</div>

<div class = "row">

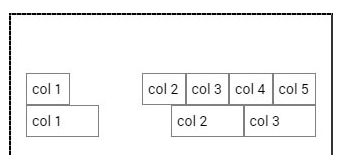
<div class = "col">col 1</div>

<div class = "col col-offset-25">col 2</div>

<div class = "col">col 3</div>

</div>

The above code will produce the following screen −



You can also vertically align the columns inside a row. There are three classes, which can be used, namely – **top**, **center** and the **bottom** class with the **col**prefix. The following code shows how to place vertically the first three columns of both rows.

**NOTE** − In the example that follows we added **“.col {height: 120px}”** to our CSS to show you the vertical placing of the columns.

<div class = "row">

<div class = "col col-top">col 1</div>

<div class = "col col-center">col 2</div>

<div class = "col col-bottom">col 3</div>

<div class = "col">col 4</div>

<div class = "col">col 5</div>

</div>

<div class = "row">

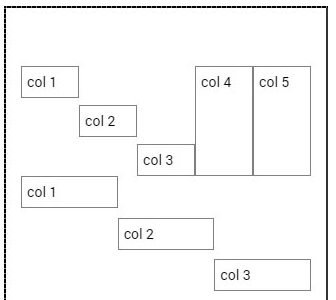
<div class = "col col-top">col 1</div>

<div class = "col col-center">col 2</div>

<div class = "col col-bottom">col 3</div>

</div>

The above code will produce the following screen −



## Responsive Grid

The Ionic Grid can also be used for a responsive layout. There are three classes available. The **responsive-sm** class will collapse columns into a single row when the viewport is smaller than a landscape phone. The **responsive-md** class will be applied when viewport is smaller than a portrait tablet. The **responsive-lg** class will be applied when viewport is smaller than a landscape tablet.

The first image after the following example shows how the **responsive-sm**class looks on a Mobile device and the second one shows how the same responsive grid looks differently on a Tablet device.

<div class = "row responsive-sm">

<div class = "col col-25">col 1</div>

<div class = "col">col 2</div>

<div class = "col">col 3</div>

<div class = "col">col 4</div>

<div class = "col">col 5</div>

</div>

<div class = "row responsive-sm">

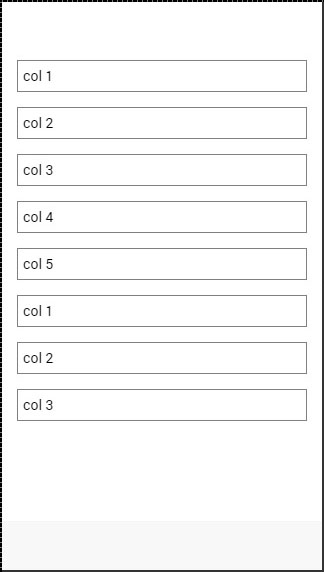
<div class = "col">col 1</div>

<div class = "col">col 2</div>

<div class = "col">col 3</div>

</div>

### Mobile Grid View



**Tablet Grid View**



**Iconic java script components**

**Action sheet**

The **Action Sheet** is an Ionic service that will trigger a slide up pane on the bottom of the screen, which you can use for various purposes.

## Using Action Sheet

In the following example, we will show you how to use the Ionic action sheet. First we will inject **$ionicActionSheet** service as a dependency to our controller, then we will create **$scope.showActionSheet()** function, and lastly we will create a button in our HTML template to call the function we created.

### Controller Code

.controller('myCtrl', function($scope, $ionicActionSheet) {

$scope.triggerActionSheet = function() {

// Show the action sheet

var showActionSheet = $ionicActionSheet.show({

buttons: [

{ text: 'Edit 1' },

{ text: 'Edit 2' }

],

destructiveText: 'Delete',

titleText: 'Action Sheet',

cancelText: 'Cancel',

cancel: function() {

// add cancel code...

},

buttonClicked: function(index) {

if(index === 0) {

// add edit 1 code

}

if(index === 1) {

// add edit 2 code

}

},

destructiveButtonClicked: function() {

// add delete code..

}

});

};

})

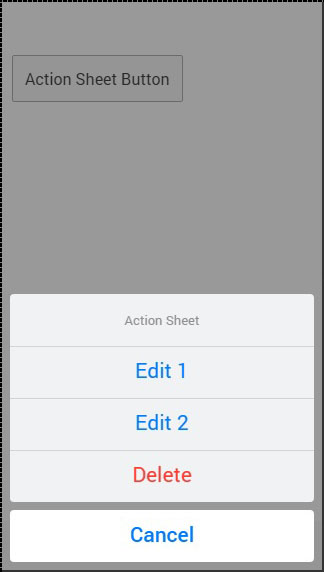
## HTML Code

<button class = "button">Action Sheet Button</button>

### Code Explained

When we tap the button, it will trigger the **$ionicActionSheet.show** function and the Action Sheet will appear. You can create your own functions that will be called when one of the options is taped. The **cancel** function will close the pane, but you can add some other behavior, which will be called when the cancel option is tapped before the pane is closed.

The **buttonClicked** function is the place where you can write the code that will be called when one of the edit options is tapped. We can keep track of multiple buttons by using the **index** parameter. The **destructiveButtonCLicked** is a function that will be triggered when the delete option is tapped. This option is **red by default**.

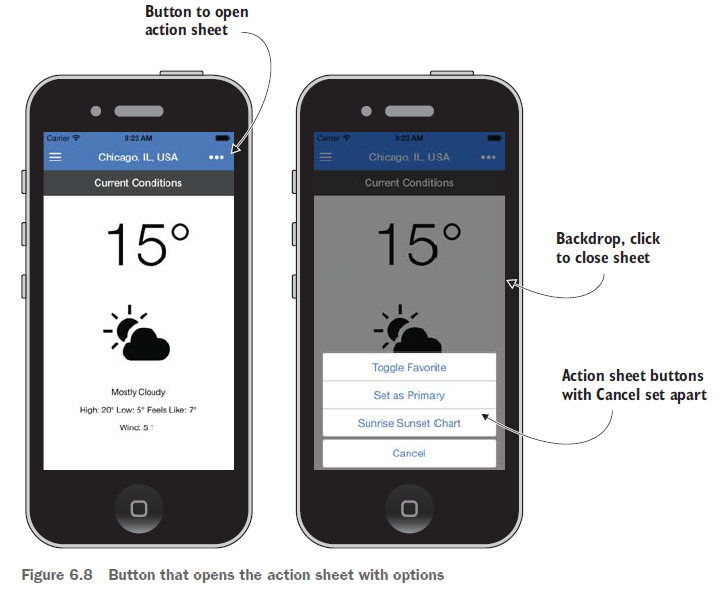


The **$ionicActionSheet.show()** method has some other useful parameters. You can check all of them in the following table.

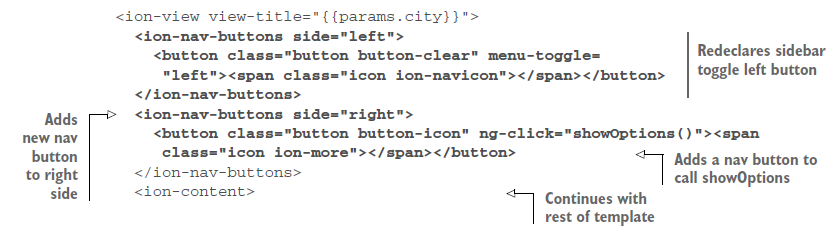
### Show Method Options

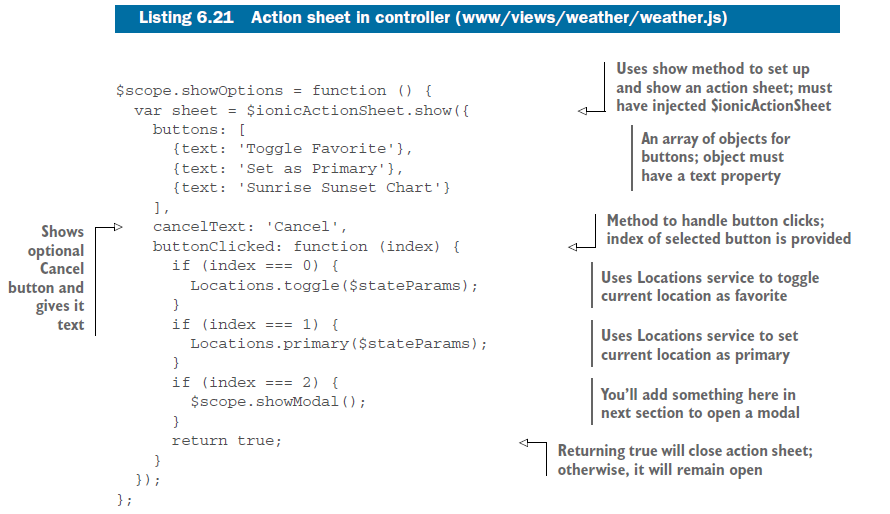
|  |  |  |
| --- | --- | --- |
| **Properties** | **Type** | **Details** |
| **buttons** | object | Creates button object with a text field. |
| **titleText** | string | The title of the action sheet. |
| **cancelText** | string | The text for cancel button. |
| **destructiveText** | string | The text for a destructive button. |
| **cancel** | function | Called when cancel button, backdrop or hardware back button is pressed. |
| **buttonClicked** | function | Called when one of the buttons is tapped. Index is used for keeping track of which button is tapped. Return true will close the action sheet. |
| **destructiveButtonClicked** | function | Called when destructive button is clicked. Return true will close the action sheet. |
| **cancelOnStateChange** | boolean | If true (default) it will cancel the action sheet when navigation state is changed. |

Another Example:



Action sheet more button (www/views/weather/weather.html)





**backdrop**

The **Ionic Backdrop** will overlay the content of the screen when applied. It will appear below other overlays (popup, loading, etc...). There are two methods that can be used for managing backdrop service. The **$ionicBackdrop.retain()** will apply backdrop over the components, and **$ionicBackdrop.release()** will remove it.

**Using Backdrop**

The following example shows how to use backdrop. We are adding **$ionicBackdrop** as a dependency to the controller, then creating the **$scope.showBackdrop()** function that will call the **retain method**immediately. Then, after three seconds, it will call the **release method**. We are using **$timeout** for the release method, so we need to add it as a controller dependency too.

.controller('myCtrl', function($scope, $ionicBackdrop, $timeout) {

$scope.showBackdrop = function() {

$ionicBackdrop.retain();

$timeout(function() {

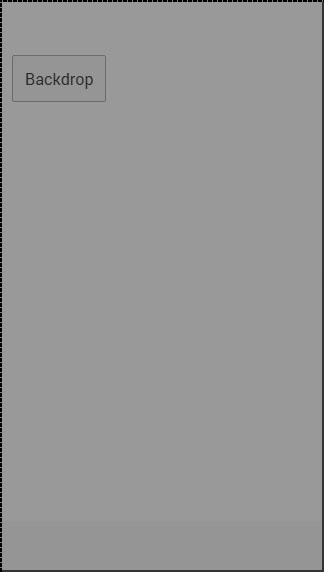
$ionicBackdrop.release();

}, 3000);

};

})

You will notice how the screen is darker in the following image, since the backdrop is applied.



**Content**

Almost every mobile app contains some fundamental elements. Usually these elements include a header and a footer, which will cover the top and the bottom part of the screen. All the other elements will be placed between these two. Ionic provide ion-content element that serves as a container, which will wrap all the other elements that we want to create.

Let us consider the following example −

<div class = "bar bar-header">

<h1 class = "title">Header</h1>

</div>

<div class = "list">

<label class = "item item-input">

<input type = "text" placeholder = "Placeholder 1" />

</label>

<label class = "item item-input">

<input type = "text" placeholder = "Placeholder 2" />

</label>

</div>

<div class = "bar bar-footer">

<h1 class = "title">Footer</h1>

</div>

**Forms**

JavaScript forms include checkbox, radio buttons and toggle.

**Using ion-checkbox**

Let us see how to use the Ionic JavaScript checkbox. Firstly, we need to create an **ion-checkbox** element in the HTML file. Inside this, we will assign an **ng-model** attribute that will be connected to the angular **$scope**. You will notice that we are using a **dot** when defining the value of a model even though it would work without it. This will allow us to keep the link between the child and the parent scopes at all times.

This is very important as it helps to avoid some issues that could happen in the future. After we create the element, we will bind its value using angular expressions.

<ion-checkbox ng-model = "checkboxModel.value1">Checkbox 1</ion-checkbox>

<ion-checkbox ng-model = "checkboxModel.value2">Checkbox 2</ion-checkbox>

<p>Checkbox 1 value is: <b>{{checkboxModel.value1}}</b></p>

<p>Checkbox 2 value is: <b>{{checkboxModel.value2}}</b></p>

Next, we need to assign values to our model inside the controller. The values we will use are **false**, since we want to start with unchecked checkboxes.

$scope.checkboxModel = {

value1 : false,

value2 : false

};

The above code will produce the following screen −



Now, when we tap the checkbox elements, it will automatically change their model value to **“true”** as shown in the following screenshot.



**Using ion-radio**

To start with, we should create three **ion-radio** elements in our HTML and assign the **ng-model** and the **ng-value** to it. After that, we will display the chosen value with angular expression. We will start by unchecking all the three radioelements, so the value will not be assigned to our screen.

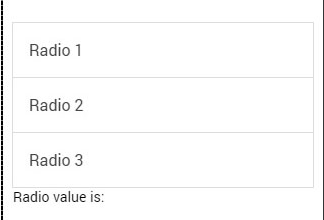
<ion-radio ng-model = "radioModel.value" ng-value = "1">Radio 1</ion-radio>

<ion-radio ng-model = "radioModel.value" ng-value = "2">Radio 2</ion-radio>

<ion-radio ng-model = "radioModel.value" ng-value = "3">Radio 3</ion-radio>

<p>Radio value is: <b>{{radioModel.value}}</b></p>

The above code will produce the following screen −



When we tap on the second checkbox element, the value will change accordingly.



**Using ion-toggle**

You will notice that toggle is similar to checkbox. We will follow the same steps as we did with our checkbox. In the HTML file, first we will create **ion-toggle**elements, then assign the **ng-model** value and then bind expression values of to our view.

<ion-toggle ng-model = "toggleModel.value1">Toggle 1</ion-toggle>

<ion-toggle ng-model = "toggleModel.value2">Toggle 2</ion-toggle>

<ion-toggle ng-model = "toggleModel.value3">Toggle 3</ion-toggle>

<p>Toggle value 1 is: <b>{{toggleModel.value1}}</b></p>

<p>Toggle value 2 is: <b>{{toggleModel.value2}}</b></p>

<p>Toggle value 3 is: <b>{{toggleModel.value3}}</b></p>

Next, we will assign values to **$scope.toggleModel** in our controller. Since, toggle uses Boolean values, we will assign **true** to the first element and **false**to the other two.

$scope.toggleModel = {

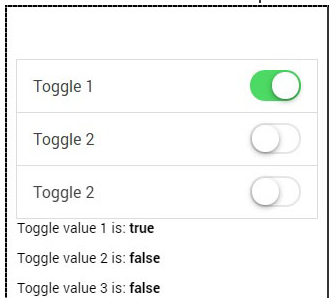
value1 : true,

value2 : false,

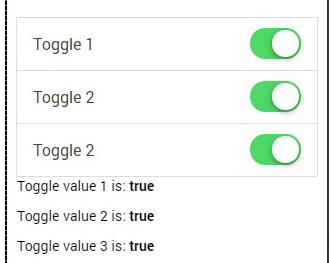
value3 : false

};

The above code will produce the following screen −



Now we will tap on second and third toggle to show you how the values change from false to true.



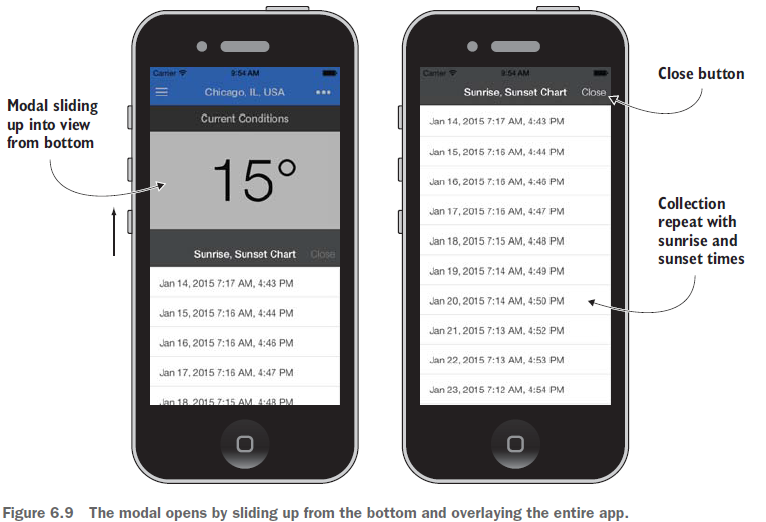
**Model**

When Ionic modal is activated, the content pane will appear on top of the regular content. Modal is basically larger popup with more functionalities. Modal will cover entire screen by default but it can be optimized the way you want.

## Using Modal

There are a two ways of implementing modal in Ionic. One way is to add separate template and the other is to add it on top of the regular HTML file, inside the **script** tags. The first thing we need to do is to connect our modal to our controller using angular dependency injection. Then we need to create a modal. We will pass in scope and add animation to our modal.

After that, we will create functions for opening, closing, destroying modal. The last two functions are placed where we can write the code that will be triggered if a modal is hidden or removed. If you do not want to trigger any functionality, when the modal is removed or hidden, you can delete the last two functions.



### Controller Code

.controller('MyController', function($scope, $ionicModal) {

$ionicModal.fromTemplateUrl('my-modal.html', {

scope: $scope,

animation: 'slide-in-up'

}).then(function(modal) {

$scope.modal = modal;

});

$scope.openModal = function() {

$scope.modal.show();

};

$scope.closeModal = function() {

$scope.modal.hide();

};

//Cleanup the modal when we're done with it!

$scope.$on('$destroy', function() {

$scope.modal.remove();

});

// Execute action on hide modal

$scope.$on('modal.hidden', function() {

// Execute action

});

// Execute action on remove modal

$scope.$on('modal.removed', function() {

// Execute action

});

});

### HTML Code

<script id = "my-modal.html" type = "text/ng-template">

<ion-modal-view>

<ion-header-bar>

<h1 class = "title">Modal Title</h1>

</ion-header-bar>

<ion-content>

<button class = "button icon icon-left ion-ios-close-outline"

ng-click = "closeModal()">Close Modal</button>

</ion-content>

</ion-modal-view>

</script>

The way we showed in the last example is when the **script** tag is used as a container to our modal inside some existing HTML file.

The second way is to create a new template file inside the **templates** folder. We will use the same code as in our last example, but we will remove the **script** tags and we also need to change **fromTemplateUrl** in controller to connect modal with new created template.

### Controller Code

.controller('MyController', function($scope, $ionicModal) {

$ionicModal.fromTemplateUrl('templates/modal-template.html', {

scope: $scope,

animation: 'slide-in-up',

}).then(function(modal) {

$scope.modal = modal;

});

$scope.openModal = function() {

$scope.modal.show();

};

$scope.closeModal = function() {

$scope.modal.hide();

};

//Cleanup the modal when we're done with it!

$scope.$on('$destroy', function() {

$scope.modal.remove();

});

// Execute action on hide modal

$scope.$on('modal.hidden', function() {

// Execute action

});

// Execute action on remove modal

$scope.$on('modal.removed', function() {

// Execute action

});

});

### HTML Code

<ion-modal-view>

<ion-header-bar>

<h1 class = "title">Modal Title</h1>

</ion-header-bar>

<ion-content>

<button class = "button icon icon-left ion-ios-close-outline"

ng-click = "closeModal()">Close Modal</button>

</ion-content>

</ion-modal-view>

The third way of using Ionic modal is by inserting HTML inline. We will use the **fromTemplate** function instead of the **fromTemplateUrl**.

### Controller Code

.controller('MyController', function($scope, $ionicModal) {

$scope.modal = $ionicModal.fromTemplate( '<ion-modal-view>' +

' <ion-header-bar>' +

'<h1 class = "title">Modal Title</h1>' +

'</ion-header-bar>' +

'<ion-content>'+

'<button class = "button icon icon-left ion-ios-close-outline"

ng-click = "closeModal()">Close Modal</button>' +

'</ion-content>' +

'</ion-modal-view>', {

scope: $scope,

animation: 'slide-in-up'

})

$scope.openModal = function() {

$scope.modal.show();

};

$scope.closeModal = function() {

$scope.modal.hide();

};

//Cleanup the modal when we're done with it!

$scope.$on('$destroy', function() {

$scope.modal.remove();

});

// Execute action on hide modal

$scope.$on('modal.hidden', function() {

// Execute action

});

// Execute action on remove modal

$scope.$on('modal.removed', function() {

// Execute action

});

});

All three examples will have the same effect. We will create a button to trigger the **$ionicModal.show()** to open modal.

### HTML Code

<button class = "button" ng-click = "openModal()"></button>

When we open modal, it will contain a button that will be used for closing it. We created this button in a HTML template.



There are also other options for modal optimization. We already showed how to use **scope** and **animation**. The following table shows other options.

|  |  |  |
| --- | --- | --- |
| **Option** | **Type** | **Detail** |
| **focusFirstInput** | boolean | It will auto focus first input of the modal. |
| **backdropClickToClose** | boolean | It will enable closing the modal when backdrop is tapped. Default value is true. |
| **hardwareBackButtonClose** | boolean | It will enable closing the modal when hardware back button is clicked. Default value is true. |

**Popover**

This is a view that will appear above the regular view.

## Using Popover

A Popover can be created by using **ion-popover-view** element. This element should be added to the HTML template and the **$ionicPopover** service needs to be injected into the controller.

There are three ways of adding popover. The first one is the **fromTemplate**method, which allows using the inline template. The second and the third way of adding popover is to use the **fromTemplateUrl** method.

Let us understand the **fromtemplate** method as explained below.

### Controller Code for Fromtemplate Method

.controller('DashCtrl', function($scope, $ionicLoading, $ionicPopover) {

// .fromTemplate() method

var template = '<ion-popover-view>' + '<ion-header-bar>' +

'<h1 class = "title">Popover Title</h1>' +

'</ion-header-bar>'+ '<ion-content>' +

'Popover Content!' + '</ion-content>' + '</ion-popover-view>';

$scope.popover = $ionicPopover.fromTemplate(template, {

scope: $scope

});

$scope.openPopover = function($event) {

$scope.popover.show($event);

};

$scope.closePopover = function() {

$scope.popover.hide();

};

//Cleanup the popover when we're done with it!

$scope.$on('$destroy', function() {

$scope.popover.remove();

});

// Execute action on hide popover

$scope.$on('popover.hidden', function() {

// Execute action

});

// Execute action on remove popover

$scope.$on('popover.removed', function() {

// Execute action

});

})

As discussed above, the second and the third way of adding popover is to use **fromTemplateUrl** method. The controller code will be the same for both ways except the **fromTemplateUrl** value.

If the HTML is added to an existing template, the URL will be the **popover.html**. If we want to place the HTML into the templates folder, then the URL will change to **templates/popover.html**.

Both examples have been explained below.

### Controller Code for the fromTemplateUrl

.controller('MyCtrl', function($scope, $ionicPopover) {

$ionicPopover.fromTemplateUrl('popover.html', {

scope: $scope

}).then(function(popover) {

$scope.popover = popover;

});

$scope.openPopover = function($event) {

$scope.popover.show($event);

};

$scope.closePopover = function() {

$scope.popover.hide();

};

//Cleanup the popover when we're done with it!

$scope.$on('$destroy', function() {

$scope.popover.remove();

});

// Execute action on hide popover

$scope.$on('popover.hidden', function() {

// Execute action

});

// Execute action on remove popover

$scope.$on('popover.removed', function() {

// Execute action

});

})

Now, we will add the **script** with template to the HTML file, which we are using for calling the popover function. **HTML code from the Existing HTML file**

<script id = "popover.html" type = "text/ng-template">

<ion-popover-view>

<ion-header-bar>

<h1 class = "title">Popover Title</h1>

</ion-header-bar>

<ion-content>

Popover Content!

</ion-content>

</ion-popover-view>

</script>

If we want to create an HTML as a separate file, we can create a new HTML file in the **templates** folder and use the same code as we used in the above-mentioned example without the **script** tags.

The newly created HTML file is as follows.

<ion-popover-view>

<ion-header-bar>

<h1 class = "title">Popover Title</h1>

</ion-header-bar>

<ion-content>

Popover Content!

</ion-content>

</ion-popover-view>

The last thing we need is to create a button that will be clicked to show the popover.

<button class = "button" ng-click = "openPopover($event)">Add Popover</button>

Whatever way we choose from above examples, the output will always be the same.



The following table shows the **$ionicPopover** methods that can be used.

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Option** | **Type** | **Detail** |
| **initialize(options)** | scope, focusFirst, backdropClickToClose, hardwareBackButtonClose | object, boolean, boolean, boolean | **Scope** is used to pass custom scope to popover. Default is the $rootScope. **focusFirstInput** is used to auto focus the first input of the popover. **backdropClickToClose**is used to close popover when clicking the backdrop. **hardwareBackButtonClose** is used to close popover when hardware back button is pressed. |
| **show($event)** | $event | promise | Resolved when popover is finished showing. |
| **hide()** | / | promise | Resolved when popover is finished hiding. |
| **remove()** | / | promise | Resolved when popover is finished removing. |
| **isShown()** | / | Boolean | Returns true if popover is shown or false if it is not. |

**Popup**

This service is used for creating a popup window on top of the regular view, which will be used for interaction with the users. There are four types of popups namely − **show**, **confirm**, **alert** and **prompt**.

## Using Show Popup

This popup is the most complex of all. To trigger popups, we need to inject the **$ionicPopup** service to our controller and then just add a method that will trigger the popup we want to use, in this case **$ionicPopup.show()**. The **onTap(e)** function can be used for adding **e.preventDefault()** method, which will keep the popup open, if there is no change applied to the input. When the popup is closed, the promised object will be resolved.

### Controller Code

.controller('MyCtrl', function($scope, $ionicPopup) {

// When button is clicked, the popup will be shown...

$scope.showPopup = function() {

$scope.data = {}

// Custom popup

var myPopup = $ionicPopup.show({

template: '<input type = "text" ng-model = "data.model">',

title: 'Title',

subTitle: 'Subtitle',

scope: $scope,

buttons: [

{ text: 'Cancel' }, {

text: '<b>Save</b>',

type: 'button-positive',

onTap: function(e) {

if (!$scope.data.model) {

//don't allow the user to close unless he enters model...

e.preventDefault();

} else {

return $scope.data.model;

}

}

}

]

});

myPopup.then(function(res) {

console.log('Tapped!', res);

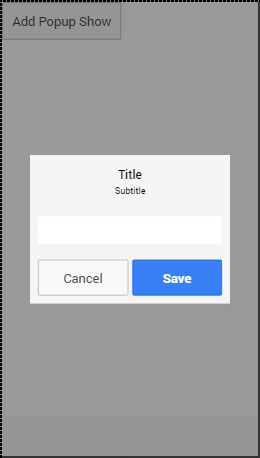
});

};

})

### HTML Code

<button class = "button" ng-click = "showPopup()">Add Popup Show</button>



You probably noticed in the above-mentioned example some new options were used. The following table will explain all of those options and their use case.

### Show Popup Options

|  |  |  |
| --- | --- | --- |
| **Option** | **Type** | **Details** |
| **template** | string | Inline HTML template of the popup. |
| **templateUrl** | string | URL of the HTML template. |
| **title** | string | The title of the popup. |
| **subTitle** | string | The subtitle of the popup. |
| **cssClass** | string | The CSS class name of the popup. |
| **scope** | Scope | A scope of the popup. |
| **buttons** | Array[Object] | Buttons that will be placed in footer of the popup. They can use their own properties and methods. **text** is displayed on top of the button, **type** is the Ionic class used for the button, **onTap** is function that will be triggered when the button is tapped. Returning a value will cause the promise to resolve with the given value. |

## Using Confirm Popup

A Confirm Popup is the simpler version of Ionic popup. It contains Cancel and OK buttons that users can press to trigger the corresponding functionality. It returns the promised object that is resolved when one of the buttons are pressed.

### Controller Code

.controller('MyCtrl', function($scope, $ionicPopup) {

// When button is clicked, the popup will be shown...

$scope.showConfirm = function() {

var confirmPopup = $ionicPopup.confirm({

title: 'Title',

template: 'Are you sure?'

});

confirmPopup.then(function(res) {

if(res) {

console.log('Sure!');

} else {

console.log('Not sure!');

}

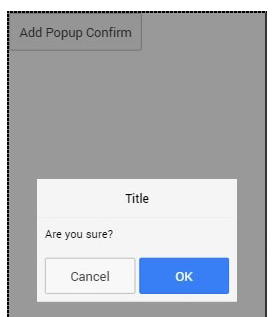
});

};

})

### HTML Code

<button class = "button" ng-click = "showConfirm()">Add Popup Confirm</button>



The following table explains the options that can be used for this popup.

### Confirm Popup Options

|  |  |  |
| --- | --- | --- |
| **Option** | **Type** | **Details** |
| **template** | string | Inline HTML template of the popup. |
| **templateUrl** | string | URL of the HTML template. |
| **title** | string | The title of the popup. |
| **subTitle** | string | The subtitle of the popup. |
| **cssClass** | string | The CSS class name of the popup. |
| **cancelText** | string | The text for the Cancel button. |
| **cancelType** | string | The Ionic button type of the Cancel button. |
| **okText** | string | The text for the OK button. |
| **okType** | string | The Ionic button type of the OK button. |

## Using Alert Popup

An Alert is a simple popup that is used for displaying the alert information to the user. It has only one button that is used to close the popup and resolve the popups’ promised object.

### Controller Code

.controller('MyCtrl', function($scope, $ionicPopup) {

$scope.showAlert = function() {

var alertPopup = $ionicPopup.alert({

title: 'Title',

template: 'Alert message'

});

alertPopup.then(function(res) {

// Custom functionality....

});

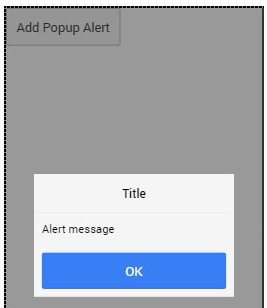
};

})

### HTML Code

<button class = "button" ng-click = "showAlert()">Add Popup Alert</button>

It will produce the following screen −



The following table shows the options that can be used for an alert popup.

### Alert Popup Options

|  |  |  |
| --- | --- | --- |
| **Option** | **Type** | **Details** |
| **template** | string | Inline HTML template of the popup. |
| **templateUrl** | string | URL of the HTML template. |
| **title** | string | The title of the popup. |
| **subTitle** | string | The subtitle of the popup. |
| **cssClass** | string | The CSS class name of the popup. |
| **okText** | string | The text for the OK button. |
| **okType** | string | The Ionic button type of the OK button. |

## Using Prompt Popup

The last Ionic popup that can be created using Ionic is **prompt**. It has an OK button that resolves promise with value from the input and Cancel button that resolves with undefined value.

### Controller Code

.controller('MyCtrl', function($scope, $ionicPopup) {

$scope.showPrompt = function() {

var promptPopup = $ionicPopup.prompt({

title: 'Title',

template: 'Template text',

inputType: 'text',

inputPlaceholder: 'Placeholder'

});

promptPopup.then(function(res) {

console.log(res);

});

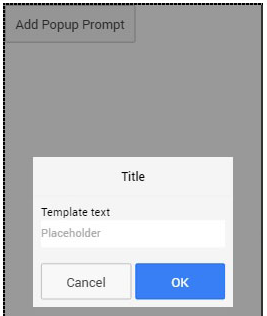
};

})

### HTML Code

<button class = "button" ng-click = "showPrompt()">Add Popup Prompt</button>

It will produce the following screen −



The following table shows options that can be used for a prompt popup.

### Prompt Popup Options

|  |  |  |
| --- | --- | --- |
| **Option** | **Type** | **Details** |
| **template** | string | Inline HTML template of the popup. |
| **templateUrl** | string | URL of the HTML template. |
| **title** | string | The title of the popup. |
| **subTitle** | string | The subtitle of the popup. |
| **cssClass** | string | The CSS class name of the popup. |
| **inputType** | string | The type for the input. |
| **inputPlaceholder** | string | A placeholder for the input. |
| **cancelText** | string | The text for the Cancel button. |
| **cancelType** | string | The Ionic button type of the Cancel button. |
| **okText** | string | The text for the OK button. |
| **okType** | string | The Ionic button type of the OK button. |

**SideMenu**

Side menu is one of the most used Ionic components. The Side menu can be opened by swiping to the left or right or by triggering the button created for that purpose.

## Using Side Menu

The first element that we need is **ion-side-menus**. This element is used for connecting the side menu with all the screens that will use it. The **ion-side-menu-content** element is where the content will be placed and the **ion-side-menu** element is the place where we can put a **side** directive. We will add the side menu to the **index.html** and place the **ion-nav-view** inside the side menu content. This way the side menu can be used throughout entire app.

### index.html

<ion-side-menus>

<ion-side-menu>side = "left">

<h1>SIde Menu</h1>

</ion-side-menu>

<ion-side-menu-content>

<ion-nav-view>

</ion-nav-view>

</ion-side-menu-content>

</ion-side-menus>

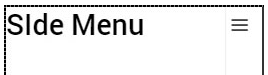
Now, we will create button **with menu-toggle = "left"** directive. This button will usually be placed in the apps header bar, but we will add it in our template file for better understanding.

When the button is tapped or when we swipe to the right, the side menu will open. You could also set the **menu-close** directive, if you would like to have one button only for closing side menu, but we will use the toggle button for this.

### HTML Template

<button menu-toggle = "left" class = "button button-icon icon ion-navicon"></button>

The above code will produce the following screen –



You can add some additional attributes to the **ion-side-menus** element. The **enable-menu-with-back-views** can be set to false to disable side menu, when the back button is showed. This will also hide the **menu-toggle** button from the header. The other attribute is **delegate-handle**, which will be used for the connection with **$ionicSideMenuDelegate**.

The **ion-side-menu-content** element can use its own attribute. When the **drag-content** attribute is set to false, it will disable the ability to open the side menu by swiping the content screen. The **edge-drag-threshold** attribute has a default value of 25. This means that swiping is allowed only 25 pixels from the left and right edge of the screen. We can change this number value or we can set it to **false** to enable swiping on the entire screen or **true** to disable it.

The **ion-side-menu** can use the **side** attribute that we showed in the example above. It will determine whether the menu should appear from the left or the right side. The **‘is-enabled’** attribute with a false value will disable the side menu, and the **width** attribute value is a number that represents how wide the side menu should be. The default value is 275.

## Side Menu Delegate

The **$ionicSideMenuDelegate** is a service used for controlling all the side menus in the app. We will show you how to use it, and then we will go through all the options available. Like all the Ionic services, we need to add it as a dependency to our controller and then use it inside the controller’s scope. Now, when we click the button, all of the side menus will open.

### Controller Code

.controller('MyCtrl', function($scope, $ionicSideMenuDelegate) {

$scope.toggleLeftSideMenu = function() {

$ionicSideMenuDelegate.toggleLeft();

};

})

### HTML Code

<button class = "button button-icon icon ion-navicon" ng-click = "toggleLeft()"></button>

The following table shows the **$ionicScrollDelegate** methods.

### Delegate Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Parameters** | **Type** | **Details** |
| **toggleLeft(parameter)** | isOpen | Boolean | Used for opening or closing side menu. |
| **toggleRight(parameter)** | isOpen | Boolean | Used for opening or closing side menu. |
| **getOpenRatio()** | / | / | Returns ratio of open part over menu width. If half of the menu is open from the left, the ration will be 0.5. If side menu is closed, it will return 0. If half of the menu is open from the right side, it will return -0.5. |
| **isOpen()** | / | Boolean | Returns true if side menu is open, false if it is closed. |
| **isOpenLeft()** | / | Boolean | Returns true if left side menu is open, false if it is closed. |
| **isOpenRight()** | / | Boolean | Returns true if right side menu is open, false if it is closed. |
| **getScrollPosition()** | / | / | Returns object with two number as properties: **left**and **right**. These numbers represent the distance the user has scrolled from the left and from the top respectively. |
| **canDragContent(parameter1)** | canDrag | Boolean | Whether the content can be dragged to open side menu. |
| **edgeDragThreshold(parameter1)** | value | Boolean|number | If the value is **true**, the side menu can be opened by dragging 25px from the edges of the screen. If it is false, dragging is disabled. We can set any number that will represent pixel value from the left and right edge of the screen. |
| **$getByHandle(parameter1)** | handle | string | Used to connect methods to the particular side menu view with the same handle.  **$ionicSideMenuDelegate. $getByHandle('my-handle').toggleLeft();** |

**SlideBox**

A Slide box contains pages that can be changed by swiping the content screen.

## Using Slide Box

The usage of the slide box is simple. You just need to add **ion-slide-box** as a container and **ion-slide** with box class inside that container. We will add height and border to our boxes for better visibility.

### HTML Code

<ion-slide-box>

<ion-slide>

<div class = "box box1">

<h1>Box 1</h1>

</div>

</ion-slide>

<ion-slide>

<div class = "box box2">

<h1>Box 2</h1>

</div>

</ion-slide>

<ion-slide>

<div class = "box box3">

<h1>Box 3</h1>

</div>

</ion-slide>

</ion-slide-box>

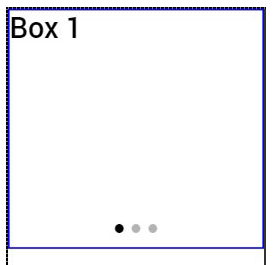
.box1, box2, box3 {

height: 300px;

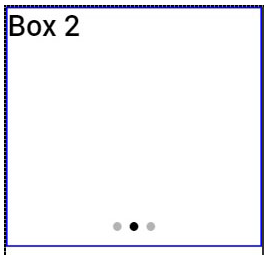
border: 2px solid blue;

}

The Output will look as shown in the following screenshot −



We can change the box by dragging the content to the right. We can also drag to the left to show the previous box.



A few attributes that can be used for controlling slide box behavior are mentioned in the following table.

### Delegate Methods

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Type** | **Details** |
| **does-continue** | Boolean | Should slide box loop when first or last box is reached. |
| **auto-play** | Boolean | Should slide box automatically slide. |
| **slide-interval** | number | Time value between auto slide changes in milliseconds. Default value is 4000. |
| **show-pager** | Boolean | Should pager be visible. |
| **pager-click** | expression | Called when a pager is tapped (if pager is visible). **$index** is used to match with different slides. |
| **on-slide-changed** | expression | Called when slide is changed. **$index** is used to match with different slides. |
| **active-slide** | expression | Used as a model to bind the current slide index to. |
| **delegate-handle** | string | Used for slide box identification with **$ionicSlideBoxDelegate**. |

## Slide Box Delegate

The **$ionicSlideBoxDelegate** is a service used for controlling all slide boxes. We need to inject it to the controller.

### Controller Code

.controller('MyCtrl', function($scope, $ionicSlideBoxDelegate) {

$scope.nextSlide = function() {

$ionicSlideBoxDelegate.next();

}

})

### HTML Code

<button class = "button button-icon icon ion-navicon" ng-click = "nextSlide()"></button>

The following table shows **$ionicSlideBoxDelegate** methods.

### Delegate Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Parameters** | **Type** | **Details** |
| **slide(parameter1, parameter2)** | to, speed | number, number | Parameter **to** represents the index to slide to. **speed** determines how fast is the change in milliseconds. |
| **enableSlide(parameter1)** | shouldEnable | boolean | Used for enambling or disabling sliding. |
| **previous(parameter1)** | speed | number | The value in miliseconds the change should take. |
| **stop()** | / | / | Used to stop the sliding. |
| **start()** | / | / | Used to start the sliding. |
| **currentIndex()** | / | number | Returns index of the curent slide. |
| **slidesCount()** | / | number | Returns total number of the slides. |
| **$getByHandle(parameter1)** | handle | string | Used to connect methods to the particular slide box with the same handle.  **$ionicSlideBoxDelegate.$getByHandle ('my-handle').start();** |

**Platform management in iconic-1**

Ionic already includes some platform-based customization, but sometimes you need your project to be even more customized. There are a few different ways you can use the platform to style an app. Learn how to implement platform classes, use JS to style your app, or use dynamic templates in the documentation below.

# Platform Classes

Ionic automagically adds classes to the <body> of your project based on the device you are using to view the project. This means if you are viewing your app on an iOS device, the <body> will have the platform-ios class applied. Some information about the different classes can be found below.

* [Platform Classes](http://ionicframework.com/docs/v1/platform-customization/platform-classes.html)
* [Styling Using AngularJS](http://ionicframework.com/docs/v1/platform-customization/styling-angularjs.html)
* [Dynamic Templates](http://ionicframework.com/docs/v1/platform-customization/dynamic-templates.html)

## Platform Device Classes

[ionic.Platform](http://ionicframework.com/docs/v1/api/utility/ionic.Platform/) is used to retrieve the device information and apply classes to the <body> of the project. These classes are used by the Ionic SDK to give the project a platform specific look and feel, and can be used by you to override styles.

|  |  |  |
| --- | --- | --- |
| **Platform** | **Class** | **Description** |
| Browser | platform-browser | Whether the application is running on the Desktop Browser - applied if you are running ionic serve |
| Cordova | platform-cordova | Whether the application is running on the device, as the device uses Cordova to display the application |
| Webview | platform-webview | Whether the application is running within a webview on the device within a native application |
| iOS | platform-ios | The device is iOS; therefore, the "look and feel" will be given the iOS treatment |
| iPad | platform-ipad | iOS device is an iPad - this class is applied, in addition to platform-ios |
| Android | platform-android | The device is Android; therefore, the "look and feel" will be given the Android treatment |
| Windows Phone | platform-windowsphone | The device is Windows Phone; therefore, the "look and feel" will be given the Windows Phone treatment |

## Platform OS Version Classes

Ionic also adds classes based on the OS version. This can be used to customize your project for a specific OS version. A class is added for the major version and the major + minor versions.

The class is created by looking at the User Agent of the OS version. The major + minor class is created by replacing the . separating the major and minor version numbers with a \_. So, if you are running your project on a device running iOS 8.4, it will add the platform-ios8and platform-ios8\_4 classes. Below are some example classes, but you are not limited to these.

|  |  |  |
| --- | --- | --- |
| Platform OS Version | Class | Description |
| iOS 8 | platform-ios8 | The OS on the device is running iOS 8 |
| iOS 8.4 | platform-ios8\_4 | The OS on the device is running iOS 8.4. This class is applied in addition to platform-ios8 |
| Android 4 | platform-android4 | The OS on the device is running Android 4 |
| Android 4.4 | platform-android4\_4 | The OS on the device is running Android 4.4. This class is applied, in addition to platform-android4 |

## Using Platform Classes to Override Styling

You can use any of the above classes to override Ionic styling. For example, if you wanted to override the header title on Android to be uppercase, you could use the following code:

**.platform-android** **.bar-header** {

text-transform: uppercase;

}

This would uppercase the title for Android, but any other device would remain lowercase.

You could even go as far as customizing the side menu style based on the platform. The possibilities are endless!

# Styling using AngularJS

The [ionic.Platform](http://ionicframework.com/docs/v1/api/utility/ionic.Platform/) utility can be used in your JavaScript controller to set the platform for your app. For the following examples, we assume you have a controller called AppCtrl encapsulating your project.

## Setting the Platform

In your AppCtrl controller, retrieve and set the platform using ionic.Platform:

.controller('AppCtrl', **function**($scope) {

$scope.platform **=** ionic.Platform.platform();

})

## Using the Platform to Dynamically Style Elements

You can then use this platform to change which elements display and what classes get added to specific elements. For example, if you’re using the tabs structure, and you want to change the way the tabs look on Android but keep the same look and feel for all other devices, you can use Angular’s ng-class to dynamically add classes based on the platform. Adding the following to the <ion-tabs> directive will add the tabs-positive class for Android platforms and the tabs-icon-top class to all devices but Android:

<ion-tabs class="tabs-stable" ng-class="{'tabs-positive': platform == 'android', 'tabs-icon-top': platform != 'android'}">

*<!-- ion-tab directives go here -->*

</ion-tabs>

You can also use Angular’s ng-attr to dynamically define attributes based on platform. For example, you may want the icons in the tabs to show on all devices except Android. You can dynamically add the icon by using the ng-attr-icon property. To define an icon for all devices minus Android on the “Home” tab, you could use the following code:

*<!-- if the platform is android don't add an icon, all other devices get an icon -->*

<ion-tab title="Home" ng-attr-icon="{{ platform != 'android' ? 'ion-home' : undefined}}" href="#/tab/home">

# Dynamically Loading Templates

Sometimes showing or changing elements based on the platform isn’t enough. There may be times when you need to use two different structures for your project, and you don’t want to place the logic in the HTML. In these cases, you can use ionic.Platform to decide which template to load in a given state. For example, the following code will load the templates/home-android.html file if the platform is Android, and templates/home.html for all other platforms:

.state('tab', {

url: "/tab",

abstract: **true**,

controller: 'AppCtrl',

templateUrl: **function**() {

**if** (ionic.Platform.isAndroid()) {

**return** "templates/home-android.html";

}

**return** "templates/home.html";

}

})

## Merges Directory

The merges directory is no longer added by default in a project, but it is still a powerful tool in customizing your app. You can create this directory at the top level of your project, alongside the www and platforms directory. From the [Cordova docs](http://cordova.apache.org/docs/en/3.5.0/guide_cli_index.md.html#The%20Command-Line%20Interface_using_merges_to_customize_each_platform):

The top-level merges directory offers a place to specify assets to deploy on specific platforms. Each platform-specific subdirectory within merges mirrors the directory structure of the www source tree, allowing you to override or add files as needed.

For example, if you wanted to have a platform-specific index.html file, you would create the following directory structure:

merges/

ios/

index.html

android/

index.html

If you wanted to have platform-specific styles or javascript, you could override those, too:

merges/

ios/

index.html

css/

platform.css

js/

app.js

android/

index.html

css/

platform.css

js/

app.js

Cordova will copy the platform specific files to the www directory in the platforms directory when you run the app. Note that you will not be able to see these differences in the browser, only when you run the app on a device or simulator.

**Plugins for iconic-1**

ngCordova comes with over 70 native Cordova plugins that you can easily add to your Angular Cordova apps. Choose the plugin you’d like to use from the menu, which will have information on which plugin you need to install and an example of how to use it in your Angular code.

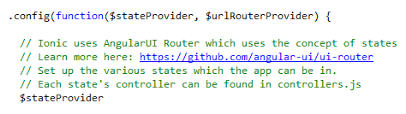
**ngCordova plugins**

some of the ngCordova plugins are : camera, touch ID, OAUTH, Push notification, geolocation, progress indicator.

stateful approach of developing iconic applications ($stateProvider.state())

 Ionic comes with the AngularUI router with it, thus we can directly use $stateProvider and $urlRouterProvider to create routes. We need to inject $stateProvider and $urlRouterProvider in our config method to make use of the AngularUI router.

We are going to build a simple two-page view, and a navigation button to navigate between the views. This is a simple example to understand syntax and setup of the router. When you open www/js/app.js, you will see a config method like below :



As you can see, $stateProvider and $urlRouterProvider are injected as dependencies to the config method. Now we will use $stateProvider to defined the states as shown below.  
 **.state('page1', {**  
**url: "/page1",**  
**templateUrl: 'templates/page1.html'**  
**})**  
**.state('page2', {**  
**url: "/page2",**  
**templateUrl: 'templates/page2.html'**  
**})**  
  
The state method of  $stateProvider are used to declare the routes. The first argument of the state method is the name of the state whereas second argument contains route configuration. Route configuration has URL and Template (rendered when URL triggered). Next we have $urlRouterProvider which provide the default URL as shown below.  
  
**$urlRouterProvider.otherwise('/page1');**  
  
Our index.html will look like below, where we are adding a back button which will be displayed when we go in any other view than the default view. We also have to tell the router which portion of the page should be updated with the contents of the state, this is done by adding **<ion-nav-view>**.  
  
<body ng-app="starter">  
  <ion-nav-bar class="bar-stable">  
    <ion-nav-back-button>  
    </ion-nav-back-button>  
  </ion-nav-bar>  
  <ion-nav-view></ion-nav-view>  
</body>  
  
The contents of page1.html are :  
  
<ion-view view-title="Simple Application Page 1">  
  <ion-content class="padding">  
    <div class="list card">  
      <div class="item item-divider">Page 1</div>  
      <div class="item item-body">  
        <button class="button button-positive" ui-sref="page2">Take me to Page 2</button>  
      </div>  
    </div>  
  </ion-content>  
</ion-view>   
  
The contents of page2.html are :  
  
<ion-view view-title="Simple Application Page 2">  
  <ion-content class="padding">  
    <div class="list card">  
      <div class="item item-divider">Page 2</div>  
      <div class="item item-body">  
        <button class="button button-positive" ui-sref="page1">Take me to Page 1</button>  
      </div>  
    </div>  
  </ion-content>  
</ion-view>

**Injection of controllers and factories in modules**

From the Angular Docs “You can think of a module as a container for the different parts of your app – controllers, services, filters, directives, etc.”

Another way to think about modules is that they are similar to namespaces in other languages where a group of classes live in a namespace. The same rule applies in Angular that in a module can live all the different pieces of an app.

## Most Basic Module

The most basic module is when you create an app with Ionic. The entire app itself is a module. Let’s look at an example.

<body ng-app="myApp">

Hello World!

</body>

angular.module('myApp', []);

Notice the value of the ng-app attribute matches the name of the module (the first parameter of the .module method).

## Module Importing

You’ll notice the second parameter is a blank array. This array is where you specify the names of the modules you would like to use in this module. It’s similar to a using or an import statement in other languages to import in other namespaces.

Let’s define another module that we’ll import into this module.

angular.module('myApp.services', [])

.factory('testFactory', function(){

return {

Hello: function(){

return "Hello World!";

}

}

});

You’ll see here we’ve defined the module as myApp.services and added a factory to the module. Another thing to notice is how there is no semicolon at the end of the first line because we are chaining the .factory method onto the .modulemethod.

Now, let’s import this services module into our first app module.

angular.module('myApp', ['myApp.services']);

All we have to do is provide the name of the module in quotes and it will import it in for us. Now, we can use the factory from the service module in our app module.

angular.module('myApp', ['myApp.services'])

.run(function(testFactory){

var hello = testFactory.Hello();

console.log(hello);

});

## Using Modules in Ionic

In the standard Ionic starter projects, there are typically 3 modules: The app module (starter), the controllers module (starter.controllers), and the services module (starter.services). There is also a fourth module that is imported: ionic. Everything related to the Ionic framework is attached to the ionic module which is imported into your app module.

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

This is a fairly standard way to organize a project by class type.

However, there is another way that is becoming quite popular and that the Ionic team themselves are looking at adopting. The idea is to organize by “feature” instead of by class types. Instead of your project having controller.js and services.js files and a templates folder with all of your .html template files, you would have a folder for each feature of your app which contains the template, controller(s), and service(s) for that feature.

.service() is a method on our module that takes a name and a function that defines the service. Pretty straight forward. Once defined, we can inject and use that particular service in other components, like controllers, directives and filters, like this:

app.controller('AppController', function (MyService) {

MyService.sayHello(); // logs 'hello'

});

Okay, clear. Now the same thing as a factory:

app.factory('MyService', function () {

return {

sayHello: function () {

console.log('hello');

}

}

});

Again, .factory() is a method on our module and it also takes a name and a function, that defines the factory. We can inject and use that thing exactly the same way we did with the service. Now what is the difference here?

Well, you might see that instead of working with this in the factory, we’re returning an object literal. Why is that? It turns out, **a service is a constructor function** whereas a factory is not. Somewhere deep inside of this Angular world, there’s this code that calls Object.create()with the service constructor function, when it gets instantiated. However, a factory function is really just a function that gets called, which is why we have to return an object explicitly.

To make that a bit more clear, we can simply take a look at the Angular source code. Here’s what the factory() function looks like:

function factory(name, factoryFn, enforce) {

return provider(name, {

$get: enforce !== false ? enforceReturnValue(name, factoryFn) : factoryFn

});

}

It takes the name and the factory function that is passed and basically returns a provider with the same name, that has a $get method which is our factory function. So what is it with this provider thing? Well, whenever you ask the injector for a specific dependency, it basically asks the corresponding provider for an instance of that service, by calling the $get()method. That’s why $get() is required, when creating providers.

In other words, if we inject MyService somewhere, what happens behind the scenes is:

MyServiceProvider.$get(); // return the instance of the service

Alright, factory functions just get called, what about the service code? Here’s another snippet:

function service(name, constructor) {

return factory(name, ['$injector', function($injector) {

return $injector.instantiate(constructor);

}]);

}

Oh look, it turns out that when we call service() it actually calls factory(). But it doesn’t just pass our service constructor function to the factory as it is. It passes a function that asks the injector to instantiate and object by the given constructor. In other words: a service calls a predefined factory, which ends up as $get() method on the corresponding provider. $injector.instantiate() is the method that ultimately calls Object.create() with the constructor function. That’s why we use this in services.

Okay, so it turns out that, no matter what we use, service() or factory(), it’s always a factory that is called which creates a provider for our service. Which brings us to the mostly asked question in the Angular history: Which one should I use?

## [Which one to use?](https://blog.thoughtram.io/angular/2015/07/07/service-vs-factory-once-and-for-all.html#which-one-to-use)

Asking that question on the internet takes us to a couple of articles and StackOverflow answers. The first is [this](https://stackoverflow.com/questions/13762228/confused-about-service-vs-factory) answer. It says:

“Basically the difference between the service and factory is as follows:”

app.service('myService', function() {

// service is just a constructor function

// that will be called with 'new'

this.sayHello = function(name) {

return "Hi " + name + "!";

};

});

app.factory('myFactory', function() {

// factory returns an object

// you can run some code before

return {

sayHello : function(name) {

return "Hi " + name + "!";

}

}

});

We now already know what happens behind the scenes, but this answer adds another comment. It says we can run code **before** we return our object literal. That basically allows us to do some configuration stuff or conditionally create an object or not, which doesn’t seem to be possible when creating a service directly, which is why **most resources recommend to use factories over services, but the reasoning is inappreciable.**

What if I told you, **we can do the exact same thing with services too?**

Yeap, correct. A service is a constructor function, however, that doesn’t prevent us from doing additional work and return object literals. In fact, constructor functions in JavaScript can return whatever they want. So we can take our service code and write it in a way that it basically does the exact same thing as our factory:

app.service('MyService', function () {

// we could do additional work here too

return {

sayHello: function () {

console.log('hello');

};

}

});

Hoppla, so what now? We just realised that, depending on how we write our services, there’s no difference between the two at all anymore.

**Icon and Splash Screen for ionic applications**

### ****Step 1)****

First we will need to create two images one for the icon and another for the splash screen. The name of the images have to be **icon.png** and **splash.png** the naming is important here so make sure you name the images as stated, the size of the icon images should be **1024×1024** and the size of the splash screen image should be **2208×2208**.

### ****Step 2)****

We now need to place the images inside the **resources** folder of our app. You will all have two images my the same name there so you need to copy and replace them.

Inside the **resources** folder there will be two folders namely and **Android** and **iOS** these will have sub-folders and images in them you do not have to worry about them.

### ****Step 3)****

Ionic by default installs the cordova splash screen plugin. So check the plugins folder and if you see a folder inside by the name “**cordova-plugin-splashscreen**” that means you have the splash screen plugin already installed, so you do not need to install it.

But for some reason you do not have the plugin install then navigate to the root of the app and run the following command

|  |  |
| --- | --- |
|  | ionic plugin add cordova-plugin-splashscreen |

This will add the splash screen plugin.

### ****Step 4)****

Now we will run the ionic resources command at the root of out app, this command will send the two images that we copies inside the **resources** folder to a remote server and will in return get icons and splash screen for all different screen sizes. You need to be connected to the internet for

So run the following command after navigating to the root of your app with the command prompt

|  |  |
| --- | --- |
|  | ionic resources |

All the images for **iOS** and **Android** for different screen sizes and orientations will be automatically copied to the respective folders inside the **resources** folder.

### ****Step 5)****

We are now ready to test our app so to install and run it on a connected device issue the following command

|  |  |
| --- | --- |
|  | ionic run android  ionic run ios |

And that’s it just like that we have splash screens for all screen sizes for **iOS** and **Android**. Notice that ionic did all the heavy lifting of re-sizing the images we just had to create two images.

**Implementation of MVC architecture (exactly as MVC of angularJS)**

An Ionic App can be broken down into 5 major pieces: Views (Made from templates), Controllers, Data (Services and Factories, App Configuration, and Directives. The Views, Controllers, and Data are the most recognizable pieces from a MVC perspective (They are exactly what you think they are), and you can probably guess what App Configuration is.

## Views

Views in an Ionic App are often referred to as templates because Angular controllers refer to them as such. Because of this, views are often stored in a /templates folder where each view is in a separate .html file. A view is where the markup for state, or page, of your app lives. In Ionic, these files tend to look something like this:

<ion-view title="About">

<ion-content>

My super cool content here!

</ion-content>

</ion-view>

The title attribute is used for other Ionic directives such as the [ion-nav-bar](http://ionicframework.com/docs/api/directive/ionNavBar/).

These views can use data binding, similar to frameworks like handlebars.

<ion-view title="About">

<ion-content>

My super cool content here! My name is {{name}}

</ion-content>

</ion-view>

{{name}} would be a variable that has been defined to the $scope in the controller.

$scope.name = "Andrew";

So the template, once data bound, would say “My super cool content here! My name is Andrew”;

These views can also use [ionic directives](http://ionicframework.com/docs/api/), angular logic (like [if statements](https://docs.angularjs.org/api/ng/directive/ngIf)), [loops](https://docs.angularjs.org/api/ng/directive/ngRepeat), and more.

## Controllers

The controllers are the brains behind the app, where the flow of logic and data is controlled. When you go to a “page” in Angular, you are really calling a controller. The controller will use a view as a template for the markup it will show to the user (hense, templates) and make calls to the data layer classes (factories/services) to get the actual data to bind to the template. The controller assigns this data to a $scope variable, which is then binded to the view. The $scope is an object that contains data defined by the controller used to build the view.

For example, I go to page site.com/#mySuperPage which will call the mySuperPage controller (mySuperPageCtrl). The controller is configured to use the superpage.html template which looks like this:

<ion-view title="About">

<ion-content>

My super cool content here! My name is {{user.name}}.

</ion-content>

</ion-view>

My controller needs to talk to the awesomeService class to get the user variable and assign it to the $scope.

.controller('MainCtrl', function($scope, $stateParams, awesomeService) {

$scope.user = awesomeService.getUser();

})

We are assuming getUser will return an object with a “name” property.

var user = {

name: "Andrew";

}

When the page is called, the controller will take the user variable that has been assigned to the $scope, and data bind it to the template. In this case, the template is calling the “name” property of the “user” $scope variable. This would result in:

<ion-view title="About">

<ion-content>

My super cool content here! My name is Andrew.

</ion-content>

</ion-view>

## Data (Factories/Services)

The Data layer of an Ionic app is the provider of data, usually from a external backend or web service. The controller requests the data from the Data layer to use in binding to the view. These Data layer classes are known as Services or Factories. The difference? [Read here](http://stackoverflow.com/questions/15666048/angular-js-service-vs-provider-vs-factory). At the end of the day, they more or less serve the same purpose. For the sake of simplicity, we’ll just refer to them both as a “data class”.

A data class generally will make some kind of http call and return a promise which when resolved contains the data received. Angular uses the [$http module](https://docs.angularjs.org/api/ng/service/$http) which will look very familiar to anyone coming from a JQuery background:

$http.get("some url here");

If you want to process the data before returning it back to the controller, you can do so when the promise is resolved and then return data that will get passed back in a promise.

$http.get("some url here").then(function(response){

//Do stuff here

return response;

});

So, a data class may look something like this:

.factory('awesomeService', function($http) {

return {

GetUser: function() {

return $http.get("some url here").then(function(response) {

//Process Stuff Here

return response;

});

},

}

})

Which when used in the controller would look something like this:

.controller('MainCtrl', function($scope, $stateParams, awesomeService) {

awesomeService.getUser().then(function(response){

$scope.user = response;

});

})

## App Configuration

The App Configuration states configures the app… thanks, captain obvious. More specifically, the routes, or states, of the app are specified, hooking up a controller to a template (view). A default route is also specified. An example .config from the [ionic-starter-tabs project](https://github.com/driftyco/ionic-starter-tabs):

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

$stateProvider

// setup an abstract state for the tabs directive

.state('tab', {

url: "/tab",

abstract: true,

templateUrl: "templates/tabs.html"

})

.state('tab.dash', {

url: '/dash',

views: {

'tab-dash': {

templateUrl: 'templates/tab-dash.html',

controller: 'DashCtrl'

}

}

})

// if none of the above states are matched, use this as the fallback

$urlRouterProvider.otherwise('/tab/dash');

});

In simple terms, the $stateProvider sets up each page or view of your app and links it with a controller.

**Login Example using MVC**

Go to your command line and start a new ionic app:

ionic start simple-login tabs

cd simple-login

ionic serve --lab

### Modify our template

As we want to display our login page before the tabs you see right now, we must add a new file in the **www/templates** folder, named login.html with this content:

<ion-view view-title="Login" name="login-view">

  <ion-content class="padding">

      <div class="list list-inset">

          <label class="item item-input">

              <input type="text" placeholder="Username" ng-model="data.username">

          </label>

          <label class="item item-input">

              <input type="password" placeholder="Password" ng-model="data.password">

          </label>

      </div>

      <button class="button button-block button-calm" ng-click="login()">Login</button>

  </ion-content>

</ion-view>

So here we have a simple view, a small login form and a button. We will assign everything needed in the controller later, for now lets add our route so we can see our view the first time. Therefore, open up the **app.js** and add this state to the already existing states:

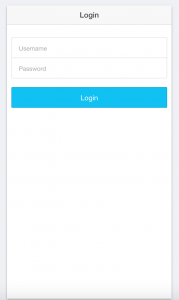
.state('login', {

      url: '/login',

      templateUrl: 'templates/login.html'

  })

You can run your app now, and you should see a view like this:

[](http://devdactic.com/wp-content/uploads/2014/12/ionic-login1.png)

### Adding the functions

BUT NOTHING WOKRS RIGHT NOW? Yeah, how shall things work without a controller? So change our state to:

.state('login', {

      url: '/login',

      templateUrl: 'templates/login.html',

      controller: 'LoginCtrl'

  })

we have to set our fallback route to /login, so that our login get’s called as first route. As Matthias said, this is not the best way for production apps, but for now include this below our states to make our app work:

$urlRouterProvider.otherwise('/login');

Now open up the **controllers.js**, and add our simple login controller:

.controller('LoginCtrl', function($scope) {

    $scope.data = {};

    $scope.login = function() {

        console.log("LOGIN user: " + $scope.data.username + " - PW: " + $scope.data.password);

    }

})

Here we create our scope variable, which is connected to the form fields we created in the login. The action is already connected to the login button, so go ahead and try to login. For now this just logs the entered data. As you often want to verify the data with a server, we will create a simple service to verify the data. Open the **services.js** and add a new service:

.service('LoginService', function($q) {

    return {

        loginUser: function(name, pw) {

            var deferred = $q.defer();

            var promise = deferred.promise;

            if (name == 'user' && pw == 'secret') {

                deferred.resolve('Welcome ' + name + '!');

            } else {

                deferred.reject('Wrong credentials.');

            }

            promise.success = function(fn) {

                promise.then(fn);

                return promise;

            }

            promise.error = function(fn) {

                promise.then(null, fn);

                return promise;

            }

            return promise;

        }

    }

})

Ok this is a bit of code, but no magic. We create our loginService with one function loginUser(), which expects the name and password of the user.

As result of our login function we return our promise, so now we must include the service in our LoginCtrl. Change our controller to this:

.controller('LoginCtrl', function($scope, LoginService, $ionicPopup, $state) {

    $scope.data = {};

    $scope.login = function() {

        LoginService.loginUser($scope.data.username, $scope.data.password).success(function(data) {

            $state.go('tab.dash');

        }).error(function(data) {

            var alertPopup = $ionicPopup.alert({

                title: 'Login failed!',

                template: 'Please check your credentials!'

            });

        });

    }

})

So here you can see how we use our service and promise correctly. First, we need to add the LoginService dependency, the $ionicPopup dependency for a simple popup and the $state for the transition to the next view. The login function now calls the loginUser function with the username and password entered in the form. Here we can use the success() and error() function for handling the login. If the credentials are as expected (user/secret), we navigate to the normal starting point of the tabs template, tab.dash, using the $state. If the credentials are wrong, we display a simple alert popup.

**http request and promises in factory()**

Promise-Based Architecture is the practice of using promises to make your code asynchronous so that things can keep happening while your app is trying to do something in the background. Using regular callbacks work for doing this, but quickly get out of hand as you get a giant Christmas tree of death.

## What is a Promise?

A promise is a guarantee that you will receive something in the future. Going with that assumption, you plan what to do when that promise is fulfilled (technically known as resolved). In other words, when I ask for something, instead of getting a response right away, I get a deferred response and I define what to do when I finally do get the actual response.

For example, say from a controller, we want a list of users from an API. Traditionally, we might have something like this:

$scope.users = UsersService.GetAll();

As stated above, because this is synchronous, our code execution must stop until we get a response back. Using promises, we will call the same method, but this time, we’ll say something along the lines of “Get me all users, **then**store them in the users variable.”

UsersService.GetAll().then(function(data){

$scope.users = data;

});

To summarize, we request something, are given a promise that we will get a response, and when we do get that response, we handle it in a then.

## $http

As covered in my article [Ionic: Using Factories and Web Services for Dynamic Data](http://mcgivery.com/ionic-using-factories-and-web-services-for-dynamic-data/), the $http methods in Angular return promises that we can handle. Basically, we request data and then define what to do when we finally do get a response back without blocking the execution of the rest of our app.

Let’s take a look at an example factory and a controller calling that factory.

.factory('userService', function($http) {

var users = [];

return {

getUsers: function(){

return $http.get("https://www.yoursite.com/users").then(function(response){

users = response;

return users;

});

}

}

})

Here, our getUsers method is going to make an http GET request. When it gets a response back, it saves the result to a private users variable, and then returns the users variable in another promise that gets returned from the getUsersmethod. In other words, make an http request, save the result, and then return the result back to the caller of the method.

However, in the mean time while we’re waiting for the http request to come back, we’re giving a promise to the caller of the getUsers method that we will eventually return something.

In our controller, we go under this assumption that we will get something back and write our then handler.

.controller('MainCtrl', function($scope, userService) {

userService.getUsers().then(function(users){

$scope.users = users;

});

})

## ngCordova

ngCordova is an Ionic run project that puts wrapper around Cordova-to-Native APIs to make them more developer friendly. One thing you’ll quickly notice going through the docs is that almost every plugin is promise based.

Take the camera for example. We want to get an image from the camera and then do something with that image. Looking at the example from the docs, we can see it uses promises to accomplish this.

$cordovaCamera.getPicture(options).then(function(imageData) {

var image = document.getElementById('myImage');

image.src = "data:image/jpeg;base64," + imageData;

});

**Themes in iconic**

Ionic was designed to be themed. You have everything you need to create a custom aesthetic across the entire set.

**Painting with broad strokes**

We define a theme as any styling that applies across our entire set. So the simplest theme would be setting Iconic in a single color. Lucky for you this only takes one simple rule.



.iconic \* {

fill: pink;

stroke: pink;

}

There, you've just colored all your icons pink. It may seem strange that we're setting all icon elements to have a fill and a stroke color. Our theme starter kit has rules that make sure that only filled elements get fill colors and stroked elements get stroke colors. This means you don't need to know whether an element is stroked or filled—just set both values and we'll figure out the rest.

**Adding detail**

One color is nice, but sometimes you'd like a little more variation. We've thought of that and have defined certain elements within our icons as accents. All accent elements are given the iconic-property-accent class. Let's say you want to add a secondary color of gray. This would also takes just one rule:



.iconic-property-accent {

fill: gray;

stroke: gray;

}

Now all of your icons that have accent elements will be colored gray. You now officially have a duo-toned icon set with two CSS rules. Not bad.

**Adding special conditions**

Another great thing about Iconic is that one icon can have many looks all within the same theme. Want to add a duo-toned active style to all your icons? Piece of cake.



.iconic-state-active \* {

fill: yellow;

stroke: yellow;

}

.iconic-state-active .iconic-property-accent {

fill: lightgray;

stroke: lightgray;

}

*We think it's a good habit to prefix all your Iconic classes with iconic. You could name your active state class whatever you'd like, but we suggest naming it like the example above.*

Now you can apply an active state to any of your icons by adding the iconic-state-active class in your HTML.

<!-- Icon with no active state -->

<img data-src="envelope.svg" class="iconic" data-state="open" alt="envelope open">

<!-- Icon with an active state -->

<img data-src="envelope.svg" class="iconic iconic-state-active" data-state="open" alt="envelope open">

**Creating touchups**

Themes will often need to give a little love to individual elements of certain icons.

Every element of each icon in Iconic has a unique class that allows it to be styled individually. This means you have fine-grained control of how your theme will be applied to Iconic. So, let's say you wanted a slightly different duo-toned look for your compass. No problemo!



.iconic-compass-needle {

fill: pink;

stroke: pink;

}

.iconic-compass-needle-axis,

.iconic-compass-direction {

fill: gray;

stroke: gray;

}

There you go, you've just officially tweaked your theme. *Remember, you can get a detailed breakdown of every class for every icon (including the*[*compass icon*](https://useiconic.com/icons/compass/)*) from our*[*icons page*](https://useiconic.com/icons)*.*