*Ionic 2 Tutorial*

1.) Transpiling- Ionic CLI transpiles code from one language to another language. We use a transpilier to convert our ES6 Code into ES5 code that is compatible with the browsers of today.

* + 1. What is ES6? ECMAScript 6 (ES6) is a standard, JavaScript is an implementation of that standard. ECMAScript defines the standard and browsers implement it.
       - 1. Classes-We can already implement object oriented code in JavaScript by using functions and prototypes, but now ES6 will provide support for class definitions and all the associated goodies like inheritance by default.

|  |  |
| --- | --- |
| ES6 | Javascript |
| class Shape {    constructor (id, x, y) {          this.id = id          this.move(x, y)      }      move (x, y) {          this.x = x          this.y = y      }  } | var Shape = function (id, x, y) {      this.id = id;      this.move(x, y);  };  Shape.prototype.move = function (x, y) {      this.x = x;      this.y = y;  }; |

* + - * 1. Modules-Allow you too modularise your code into packages that can be imported anywhere you need in your application

|  |  |
| --- | --- |
| ES6 | Javascript |
| // lib/math.js  export function sum (x, y) { return x + y }  export var pi = 3.141593    // someApp.js  import \* as math from "lib/math"  console.log("2π = " + math.sum(math.pi, math.pi))    // otherApp.js  import { sum, pi } from "lib/math"  console.log("2π = " + sum(pi, pi)) | // lib/math.js  LibMath = {};  LibMath.sum = function (x, y) { return x + y };  LibMath.pi = 3.141593;    // someApp.js  var math = LibMath;  console.log("2π = " + math.sum(math.pi, math.pi));    // otherApp.js  var sum = LibMath.sum, pi = LibMath.pi;  console.log("2π = " + sum(pi, pi)); |

* + - * 1. Promises-Provide a much nicer format for grabbing asynchronous data (e.g. timeouts; data you need to wait for when you fetch something from a server or device)

|  |  |
| --- | --- |
| ES6 | Javascript |
| function msgAfterTimeout (msg, who, timeout) {  return new Promise((resolve, reject) => {  setTimeout(() => resolve(`${msg} Hello ${who}!`), timeout)  })  }  msgAfterTimeout("", "Foo", 100).then((msg) =>  msgAfterTimeout(msg, "Bar", 200)  ).then((msg) => {  console.log(`done after 300ms:${msg}`)  }) | function msgAfterTimeout (msg, who, timeout, onDone) {      setTimeout(function () {          onDone(msg + " Hello " + who + "!");      }, timeout);  }  msgAfterTimeout("", "Foo", 100, function (msg) {      msgAfterTimeout(msg, "Bar", 200, function (msg) {          console.log("done after 300ms:" + msg);      });  }); |

* + - * 1. Block Scoping-Allow you to use the new let keyword to define a variable only within a single block of code

|  |  |
| --- | --- |
| ES6 | Javascript |
| for (let i = 0; i < a.length; i++) {  let x = a[i]  //etc.  } | When you define a variable in JavaScript it is available anywhere within the function that I defined it in. The new block scoping features in ES6 allow you to use the new let keyword to define a variable only within a single block of code like this: |

* + 1. What is Typescript?
       - 1. Typescript is defined as a typed superset of Javacript that compiles to plain Javascript; basically adds typing, classes, and interfaces to Javascript.
         2. TypeScript still provides the ability to use static typing in JavaScript (which means it is evaluated at compile time, opposed to dynamic typing which is evaluated at run time).

|  |  |
| --- | --- |
| Typescript (.ts) | Javascript (.js) |
| class Greeter {  greeting: string;  constructor (message: string) {  this.greeting = message;  }  greet() {  return "Hello, " + this.greeting;  }  } | var Greeter = (function () {  function Greeter(message) {  this.greeting = message;  }  Greeter.prototype.greet = function () {  return "Hello, " + this.greeting;  };  return Greeter;  })(); |

* 1. Ionic 2 Syntax
     1. Web Components- a new standard on the web to create modular, self-contained, pieces of code that can easily be inserted into a web page; like Widgets in WordPress.

|  |  |
| --- | --- |
| Ionic 1 | Ionic 2 |
| <div id="slider">  <input checked="" type="radio" name="slider" id="slide1" selected="false">  <input type="radio" name="slider" id="slide2" selected="false">  <input type="radio" name="slider" id="slide3" selected="false">  <input type="radio" name="slider" id="slide4" selected="false">  <div id="slides">  <div id="overflow">  <div class="inner">  <img src="images//rock.jpg">  <img src="images/grooves.jpg">  <img src="images/arch.jpg">  <img src="images/sunset.jpg">  </div>  </div>  </div>  <label for="slide1"></label>  <label for="slide2"></label>  <label for="slide3"></label>  <label for="slide4"></label>  </div> | <img-slider>  <img src="images/sunset.jpg" alt="a dramatic sunset">  <img src="images/arch.jpg" alt="a rock arch">  <img src="images/grooves.jpg" alt="some neat grooves">  <img src="images/rock.jpg" alt="an interesting rock">  </img-slider> |

* + 1. Angular 2 Syntax
       - 1. Attribute

|  |
| --- |
| Attribute- some data you supply to the element. |
| <input id="myInput" value= "Hey there"> |

* + - * 1. Property

|  |
| --- |
| Property-accessed and modified as an object |
| var myInput = document.getElementById('myInput');  console.log(myInput.value);  myInput.value = "What's up?";  console.log(myInput.value);  myInput.value = new Object(); |

* + - * 1. Method

|  |
| --- |
| Method- a function that we can call on the element. |
| myInput.setValue(‘Hello’); |

* + - * 1. Angular Binding

Note that firstName is an expression, not a string. This means that the value of the firstName variable (defined in your TypeScript file) will be used here, not the literal string ‘firstName’.

|  |
| --- |
| Binding a Property to a Value |
| <input [value]="firstName"> |

This will call the someFunction function and pass in the event whenever the button is clicked. You can replace click with any native or custom event you like.

|  |
| --- |
| Binding a Function to an Event |
| <button (click)="someFunction($event)"> |

Rendering Expressions with Interpolations- allows you to render out expressions like variables and mathematical calculations into your template as a string

|  |
| --- |
| Interpolations |
| <p>Sum: {{1 + 1}}</p>  Output- Sum: 2 |

Two way Data Binding- allows you to set up variables that will remain in sync no matter where they are changed. If the variable is updated in the class it will be reflected immediately in the template, if the variable is updated in the template it will be immediately reflected in the class. This sets the value to the expression name and when we detect the input event we update name to be the new value that was entered. To make this easier, we can still use ng-model in Angular like this to achieve the same thing

|  |  |
| --- | --- |
| Option 1 | Option 2 |
| <input [value]="name" (input)="name = $event.target.value"> | <input [(ngModel)]="name"> |

Template Variable to Access Elements- This creates a local variable that we can use to access the element.

|  |  |
| --- | --- |
| Create Template Variable | Access Element |
| <p #myParagraph></p> | <button (click)="myParagraph.innerHTML = 'Once upon a time...'"> |

* + 1. Structural Directives- We can use structural directives to modify our templates. The \*ngIf directive will remove a DOM element if the condition it is attached to is not met. The \*ngFor directive can loop over an array, and repeat a DOM element for each element in that array.

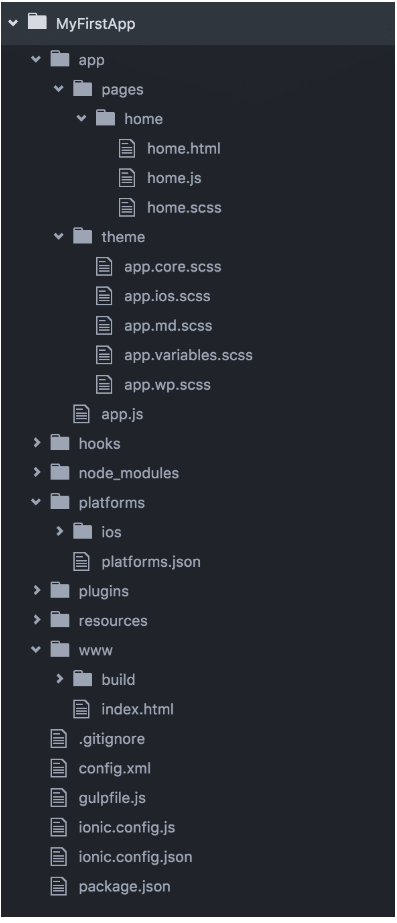
|  |
| --- |
| Directives |
| <section \*ngIf="showSection">  <li \*ngFor="let item of items"> |

* + 1. Annotations-Annotations like @Component, @Directive and so on allow you to attach information to your components.

|  |
| --- |
| Annotations |
| @Component({  selector: 'my-component',  services: [MyService]  }) |

* + 1. Import & Export: It’s a similar concept to Dependency Injection in Angular 1, where we would inject services into the controller.

|  |  |
| --- | --- |
| Annotations | Inonic 1 Injections |
| import { Component } from '@angular/core';  import { NavController } from 'ionic-angular';    @Component({  templateUrl: 'hello-ionic.html'  })  export class HelloIonicPage {    constructor(private navCtrl: NavController) {    }  }  import { HelloIonicPage } from './pages/hello-ionic/hello-ionic'; //Access page | .controller('ExampleCtrl', function($scope, $state, $myService) { |

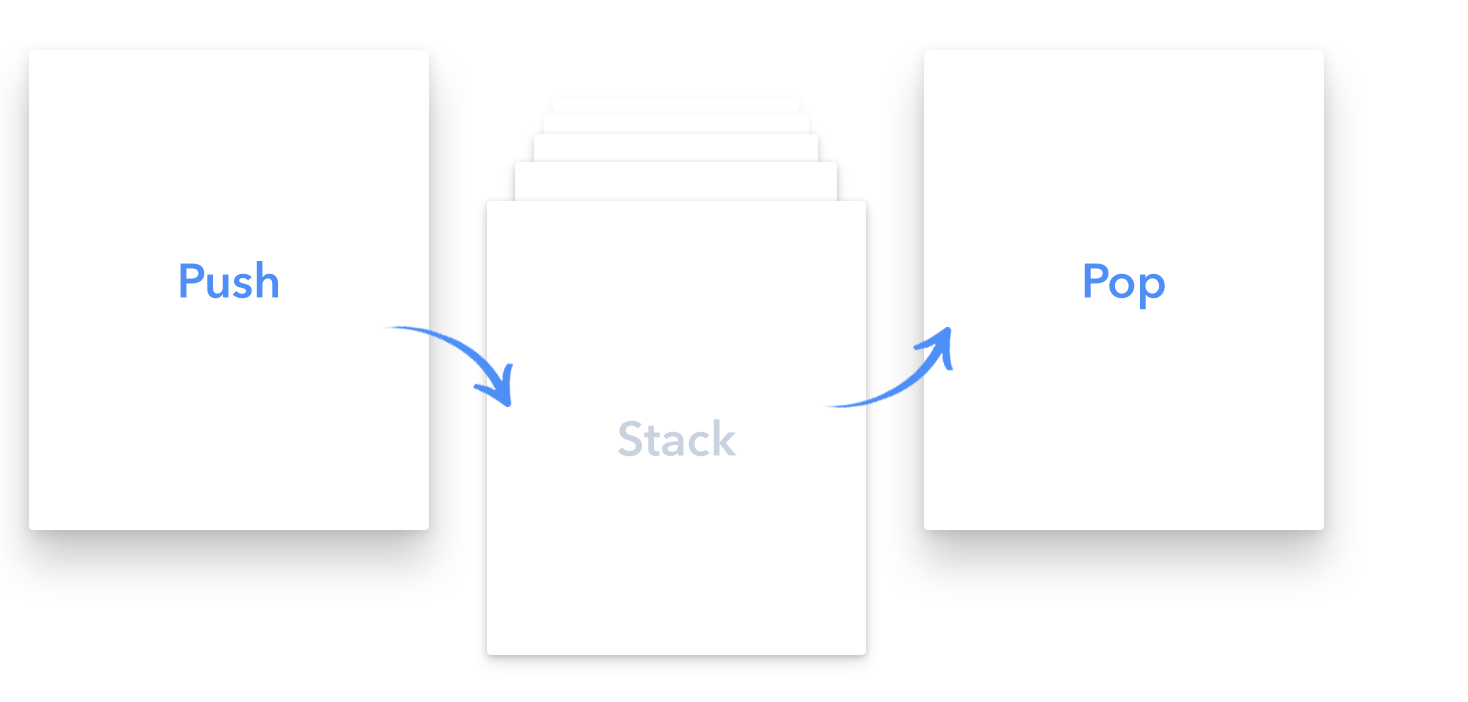
1. Navigation & File Structure
   1. Folders
      1. app – App: Starting with folders, we can see that the bulk of our app gets implemented here: pages, services, etc.
         * + pages – The ininital structure you can find in a folder app/pages/home contains three files: home.ts, home.html, and home.scss. Home.js contains implementation of the corresponding class HomePage component.
           + themes – This folder will contain all SCSS files which contain styling on application level.
           + app.js – This file is containing the code of the root component of our application. As each Angular 2 application is built up of a tree of components the root component is the top component and the starting point of the application. This is where we bootstrap our application and where you will find your app’s @App decorator. There are a handful of global settings you’ll want to pass in. Most importantly, you’ll want to pass your app’s root template, config and list of providers.
      2. hooks – Hooks: custom scripts which are executed as part of the Cordova build process. Of course, hooks are an advanced feature and for most cases not needed. But if you want to add custom logic to the build process you could create hooks and place the corresponding scripts in this directory.
      3. node\_modules – This is the storage location where you can find all installed NPM dependencies. If you add another dependency to package.json and afterwards execute the npm install command the package is automatically downloaded and installed into a new subfolder within node\_modules. As the Ionic 2 framework is also installed via NPM package, this folder does also contain a subfolder named ionic-angular where you can find the Ionic 2 framework files.
      4. platforms- Contains your platform targets like iOs or Android.
      5. plugins- First, you will need to install the plugin you want to use. Then you will need to import the plugin into the component that you want to use it in. Let’s add it to the automatically generated home page in home.ts:

|  |  |
| --- | --- |
| Geolocation Plugin | Home.ts |
| * + ionic plugin add cordova-plugin-geolocation | import { Component } from '@angular/core';  import { Platform } from 'ionic-angular';  import { Geolocation } from 'ionic-native';  import { HomePage } from '../pages/home/home';    @Component({  template: '<ion-nav [root]="rootPage"></ion-nav>'  })  export class MyApp {    rootPage: any = HomePage;    constructor(platform: Platform) {    platform.ready().then(() => {    });  }  } |

* + 1. resources-The resources folder contains images which are used for the splash screen and icons of your app when building for iOS and Android.
    2. www-This is where your index.html file lives. This is the web root folder containing the compiled code of your Ionic 2 application for the various platforms. Inside of the www folder you’ll find the subfolder build were compiled code is stored.
       - * pages – The ininital structure you can find in a folder app/pages/home contains three files: home.js, home.html, and home.scss. Home.js contains implementation of the corresponding class HomePage component.
         * themes: This folder will contain all SCSS files which contain styling on application level.
    3. www/index.html- This file is stored in www directly and is the starting point of our application. As you can see in the following listing index.html is used to include important scripts which we make use of in our Ionic 2 application.This is obviously where you should update your meta tags and add in any required scripts (cordova, polyfills, vendor build, app build, etc.)

|  |
| --- |
| Index.html |
| <!DOCTYPE html>  <html lang="en" dir="ltr">  <head>    <title>Ionic</title>    <meta charset="UTF-8">    <meta name="viewport" content="width=device-width, initial-scale=1.0, minimum-scale=1.0, maximum-scale=1.0, user-scalable=no">    <meta name="format-detection" content="telephone=no">    <meta name="msapplication-tap-highlight" content="no">      <link ios-href="build/css/app.ios.css" rel="stylesheet">    <link md-href="build/css/app.md.css" rel="stylesheet">    <link wp-href="build/css/app.wp.css" rel="stylesheet">  </head>    <body>    <ion-app></ion-app> //This component initilaizes the application.      <!-- cordova.js required for cordova apps -->    <script src="cordova.js"></script>    <!-- Polyfill needed for platforms without Promise and Collection support -->    <script src="build/js/es6-shim.min.js"></script>    <!-- Zone.js and Reflect-metadata  -->    <script src="build/js/Reflect.js"></script>    <script src="build/js/zone.js"></script>    <!-- the bundle which is built from the app's source code -->    <script src="build/js/app.bundle.js"></script>  </body>    </html> |

* + 1. Configuration Files

1. package.json- This file contains all dependencies (NPM packages) of our application. You can add new packages or update the version of packages already included. By executing the command npm install in the project directory the dependencies listed in package.json are downloaded and added to the project automatically.
2. config.xml - This configuration file is used by Cordova and includes settings which are relevant for building your application for the iOS and Android platform.
3. gulpfile.js- Here you find hooks for modifying the Ionic gulp tasks. Use these to modify the Ionic build. There are a number of custom hooks you can take advantage of. You most likely will not need to modify this file.
4. webpack.config.js (optional)- If you are using webpack for builds then you certainly will need this for adjusting your webpack build settings.
5. ionic.config.json-These are your project-specific settings. Primarily it’s just the app name and id along with any proxies you may need for your app.
   1. Navigation- An important part of all this is the NavController. It is imported by default. A reference to the NavController is created so that we can use it anywhere within the class. 
      1. Root Page- The root page is different to the root component, typically the root component (which is defined in app.component.ts) will declare what the root page is – the root page can be changed throughout the application, the root component can not. If the view you are switching to is not a child of the current view, or it is a different section of the application, then you should instead change the root page. For example, if you have a login screen that leads to the main application you should change the root page to be your main logged in view once the user has successfully authenticated. If you have a side menu with the options Dashboard, Shop, About and Contact you should set the root page to whichever of these the user selects.

|  |  |
| --- | --- |
| Import NavController | Inject into Constructor |
| import { NavController } from 'ionic-angular'; | @Component({  templateUrl: 'home.html',  })  export class MyPage {  constructor(public navCtrl: NavController) {    }  } |

|  |  |  |
| --- | --- | --- |
| Root Page | | App.ts |
| this.navCtrl.setRoot(MenuPage);  this.navCtrl.popToRoot(); | rootPage: any = MenuPage;  //Tells the nav to drop all the views pushed to the queue all the way to the root.  if (authed) {  this.rootPage = MenuPage;  } else {  this.rootPage = LoginPage;  } | |

* + 1. Push-To push a page, that means to make it the current page. If you want the ability to navigate back to the previous view from the new view, you should push. This uses the reference to the NavController we created before, and all you need to supply to it is a reference to the component that you want to navigate to, which you will need to make sure you also import at the top of the file

|  |  |  |
| --- | --- | --- |
| Push | | Import Page |
| this.navCtrl.push(SecondPage); | import { SecondPage } from '../second-page/second-page'; | |

* + 1. Pop- To pop a page, that means to return to previous page. If you want the ability to navigate back to the previous view from the child view, you should push. When you push a page, a ‘Back’ button will automatically be added to the nav bar, so you often don’t need to worry about using pop to navigate back to the previous page since the ‘Back’ button does this automatically for you.

|  |
| --- |
| Pop |
| this.navCtrl.pop();  this.navCtrl.popToRoot(); |

* + 1. Passing Data Between Pages- NavParams is used to pass data between pages. Data must be passed with a push call (or setRoot).

|  |  |  |
| --- | --- | --- |
| Step 1-First, you must pass through the data you want within the push call. | Step 2-There is an extra parameter which is an object that contains the data we want to send through to SecondPage. Then on the receiving page we need to import NavParams and inject it into our constructor | Step 3-Grab the data that was passed through |
| this.navCtrl.push(SecondPage, {      thing1: data1,      thing2: data2  }); | import { Component } from '@angular/core';  import { NavController, NavParams } from 'ionic-angular';    @Component({  templateUrl: 'second-page.html'  })  export class SecondPage {  constructor(public navCtrl: NavController, public navParams: NavParams){  }  } | this.navParams.get('thing1');  this.navParams.get(‘thing2’); |

1. Getting Started
   1. Install Node & Ionic-

|  |
| --- |
| Ionic Setup CLI |
| // Download the installer for Node.js 6 or greater   * + npm install –g cordova ionic@beta |

* 1. Creating your app

|  |  |
| --- | --- |
| Ionic Start CLI | Index.html |
| * + ionic start helloWorld tabs --v2 --ts   + cd helloWorld   + ionic platform add android   + cordova plugin add cordova-plugin-whitelist   //In index.html add: | <meta http-equiv="Content-Security-Policy" content="script-src 'self' 'unsafe-eval' 'unsafe-inline' \*; object-src 'self'; style-src 'self' 'unsafe-inline'; media-src \*"> |

* 1. Working with pages- Each page in an Ionic 2 application is a component, and every component has a class definition in a TypeScript file, and an HTML template. The step above will create your first @Page of your app inside app/pages/home.
     1. Generated Home Page

|  |  |
| --- | --- |
| Home @Page | Home Files |
|  | //home.html  //home.ts: The TypeScript for your page, where the @Page component is defined  //home.scss: a file to put any custom SASS styles for this page |

* + 1. Assign a Root Page-This is a special way AngularJS 2 allows us to access navigation directly from application classes. In this case, an application is referencing root property to rootPage variable. Sets root to movie list page.

|  |  |
| --- | --- |
| Home @Page | Home.ts |
| <ion-nav [root]="rootPage"></ion-nav> | import {Component} from '@angular/core';  import {Platform, ionicBootstrap} from 'ionic-angular';  import {StatusBar} from 'ionic-native';  import {MovieListPage} from './pages/movie-list/movie-list';    @Component({  templateUrl: 'build/home.html',  })  export class MyApp {  rootPage: any = MovieListPage;  constructor(platform: Platform) {  platform.ready().then(() => {  StatusBar.styleDefault();  });  }  }    ionicBootstrap(MyApp); |

* + 1. Adding a New page manually

|  |  |
| --- | --- |
| About @Page | About Command |
|  | * + ionic g page about   // Generates about.html, about.ts, and about.scss in  app/pages/about |

* 1. Ionic Serve

|  |
| --- |
| Ionic Serve CLI |
| // View our app in browser   * + ionic serve |

* 1. Assigning a Value in class HomePage

|  |  |
| --- | --- |
| Home.html @Page | Home.ts |
| <ion-navbar \*navbar>  <ion-title>  Home  </ion-title>  </ion-navbar>  <ion-content class="home">  <ion-card>  <ion-card-content>  Hello {{name}}  </ion-card-content>  </ion-card>  </ion-content> | import {Page} from 'ionic-angular';  @Page({  templateUrl: 'build/pages/home/home.html'  })  export class HomePage {  public name;    constructor(){  this.name = "Andrew";  }  } // Outputs Hello Andrew |

* 1. Navigating From Home To About -

Step 1: To navigate from our home page to our about page, we will need to import our AboutPage class into our home.ts file for use in our HomePage class. Next, we should add a constructor to our HomePage class and assign our AboutPage to a property so we can use it in our template.

Step 2: Next in our home template, we can add a button with NavPush and pass our aboutPage property.

|  |  |  |  |
| --- | --- | --- | --- |
| Home.ts | Home.html | About.html | Browser (Ionic Serve) |
| import {Page} from 'ionic-angular';  import {AboutPage} from '../about/about';  @Page({  templateUrl: 'build/pages/home/home.html'  })  export class HomePage {  aboutPage = AboutPage;  constructor(){  }  } | <ion-card-content>  Hello World  <button [navPush]="aboutPage">  Go To About  </button>  </ion-card-content> | <ion-content padding class="about">  This is my super awesome about page.  </ion-content> |  |

Step 3: And we’ll add some content to our about template.