

MOBILE DEVELOPMENT FOR WEB DEVELOPERS



Joshua Morony

WELCOME TO THE COURSE!

NOTE: This is a preview version of Mobile Development for Web Developers. It includes the first lessons from both Section 3 and Section 4. If you would like to purchase the entire course, please visit: <https://joshmorony.com/mobile-development-for-web-developers/>

Hello and welcome to **Mobile Development for Web Developers**! This course is all about taking your existing skills as a web developer, and using them to start your journey in developing mobile applications.

People come into this course with varying degrees of experience, but whatever your level is shouldn't matter too much. All of the lessons go into a lot of detail, I only expect that you:

- Have a **solid understanding** of **HTML** and **CSS**
- Have at least **introductory experience** with **JavaScript**
- Understand the **basics of programming** (for loops, if / else, while etc.)

If you have a strong web development background then this probably won't be a problem for you. Some people may not have as much experience with JavaScript and programming though so if you need a bit of a crash course before we get started I would recommend taking a look at:

- [JavaScript at Codecademy](#)

and no I would recommend the following quick tutorial if you need to brush up on HTML and CSS:

- [HTML5 and CSS3 techniques](#)

If you are rusty in any of these areas, but you want to get into building mobile apps right away, then you will pick up most of what you need to know from this course anyway (it might just take a little longer). But I **highly recommend** you make it a point to go back and study the basics of JavaScript at some point, this course is not intended as an intro to JavaScript and it will be so much easier to improve your knowledge in the future if you understand the basics.

How is the course structured?

Before we get started, I want to talk a little bit about what you can expect from this course. The course is split into 5 sections:

- **Section 1:** Introduction to HTML5 Mobile Development
- **Section 2:** Native Wrappers
- **Section 3:** Mobile App Development with Ionic
- **Section 4:** Mobile Game Development with Phaser
- **Section 5:** App Store Submission

Depending on which package you purchased, you will either have access to all of these sections, or just those specific to mobile app development or mobile game development.

NOTE: *If you purchased one of the smaller packages and want to upgrade to the full course, you can use the links I would have sent to you via email to get the entire course discounted to the same price you would have paid if you had bought the full package initially. If you can't find the email, feel free to get in touch and I'll send it to you!*

Each section contains a series of lessons that focus on the topic of that section. A lesson will usually consist of a **discussion or tutorial**, and depending on the lesson it may also be accompanied with **source code and other resources**.

At the end of each tutorial style lesson, you will be challenged with a **homework task**. These are **optional** and extend on concepts we would have worked on in the lesson. If you choose not to do these you will still be able to finish the applications, but completing tasks that I do not walk you through will do a lot to help cement your knowledge. If you're not confident in completing them, you can always come back later to finish the homework tasks.

Remember that all general resources for this course and resources specific to lessons will be **stored in the download pack** you received - make sure to reference this as needed. If you ever find yourself stuck, referencing the source code for the lesson you are up to is a great way to figure out what went wrong.

How fast should you complete the course?

The course is completely self paced so it's up to you to determine how quickly you want to work your way through the content - this is going to depend largely on your current skill level.

Although the source code is provided, I would encourage you to code along as you go as it will help you retain more of what you learn. If you try to rush through the course by just copy and pasting code, you probably won't have actually learned much by the end. On the other hand, if you take too long to complete the course you may find your motivation lapse and you may not even end up finishing it (and I really think that you should!). As a recommendation, **I would suggest consistently completing 1-3 lessons each day until the course is complete.** Try reading through all of Section 1 & 2 today though, these are theory lessons that don't involve any actual coding so you should be able to get through them pretty quickly.

Conventions used in this book

The layout used in this book doesn't really need much explaining, however you should look out for:

> Blocks of text that look like this

As they are actions you have to perform. For example these blocks of text might tell you to create a file or make some code change.

NOTE: *You will also come across blocks of text like this. These will contain little bits of information that are related to what you are currently doing.*

IMPORTANT: *You will also see a few of these. These are important "Gotchas" you should pay careful attention to.*

Ok, enough chat! Let's get into the first section: **Introduction to HTML5 Mobile Development.** Good luck and have fun!

SECTION 1

INTRODUCTION TO HTML5 MOBILE DEVELOPMENT

Lesson 1

NOT AVAILABLE IN PREVIEW

NOTE: This is a preview version of Mobile Development for Web Developers. It includes the first lessons from both Section 3 and Section 4. If you would like to purchase the entire course, please visit: <https://joshmorony.com/mobile-development-for-web-developers/>

SECTION 2

NATIVE WRAPPERS

Lesson 1

NOT AVAILABLE IN PREVIEW

NOTE: This is a preview version of Mobile Development for Web Developers. It includes the first lessons from both Section 3 and Section 4. If you would like to purchase the entire course, please visit: <https://joshmorony.com/mobile-development-for-web-developers/>

SECTION 3

MOBILE APP DEVELOPMENT

Lesson 1

GETTING STARTED WITH IONIC

By now you should have a decent understanding of what a hybrid HTML5 mobile application is and how it's different to a normal native application. We've also discussed some specific frameworks that you can use to build these mobile applications like [Ionic](#), [Sencha Touch](#), [jQuery Mobile](#) and so on.

Now we are going to start getting into the practical side of things and building our own application. We will be using **Ionic** with **Cordova** / **PhoneGap** to build the application. The reason for choosing Ionic is that it has a reasonably small learning curve, it is flexible and powerful, and it is probably the most popular HTML5 mobile framework in use today.

To reiterate the different roles Ionic and Cordova serve: Ionic is used to build the majority of the application including things like the user interface, screen transitions, logic and data storage. Cordova is used to access native device API's (like the Camera) and to wrap the application up as a native application so that it can be submitted to App Stores.

Getting Started with Ionic

Before we can start building an application with Ionic we need to get everything set up on our computer first. It doesn't matter if you have a Mac or PC, you will still be able to finish this course and produce both an iOS and Android application that is ready to be submitted to app stores.

First you will need to install **Node.js** on your machine. Node.js is a platform for building fast, scalable network applications and it can be used to do a lot of different things. Don't worry if you're not familiar with it though, we won't really be using it much at all - we need it installed for Ionic to run properly but we barely have to do anything with it.

Installing Ionic and Cordova

> Visit the following website to install Node.js:

<https://nodejs.org/>

Once you have Node.js installed, you will be able to access the **node package manager** or **npm** through your command terminal.

> Install Ionic and Cordova by running the following command:

```
npm install -g cordova ionic
```

You should also set up the Android SDK on your machine by following this guide:

<http://ionicframework.com/docs/ionic-cli-faq/#android-sdk>

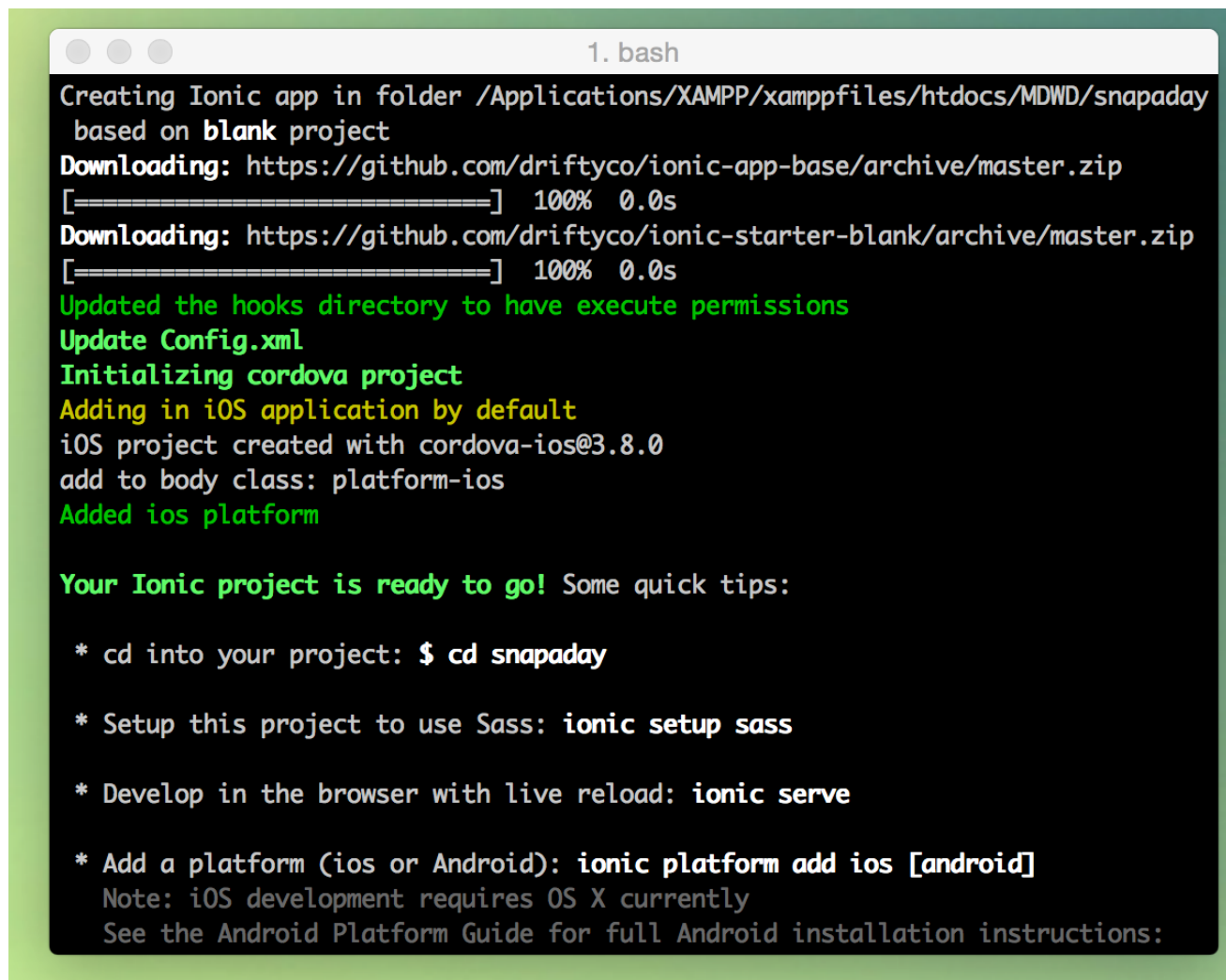
You don't have to worry about setting up the iOS SDK as if you have a Mac it will be done automatically and if you don't have a Mac then you can't set it up on your computer anyway.

Generating an application

Once Ionic is installed, generating applications is *really* easy. You can simply run the `ionic start` command to create a new application with all of the boilerplate code and files you need.

> Run the following command to generate a new Ionic application:

```
ionic start snapaday blank
```

A terminal window titled '1. bash' showing the output of the 'ionic start snapaday blank' command. The output indicates the creation of an Ionic app in the folder '/Applications/XAMPP/xamppfiles/htdocs/MDWD/snapaday' based on the 'blank' project. It shows two download steps for 'ionic-app-base' and 'ionic-starter-blank' archives, both completed at 100% in 0.0s. Subsequent steps include updating the hooks directory, updating Config.xml, initializing the cordova project, and adding the iOS application by default. The terminal concludes with a message 'Your Ionic project is ready to go!' and provides several quick tips for development, such as using 'cd snapaday', 'ionic setup sass', 'ionic serve', and 'ionic platform add ios [android]'.

```
1. bash
Creating Ionic app in folder /Applications/XAMPP/xamppfiles/htdocs/MDWD/snapaday
based on blank project
Downloading: https://github.com/driftyco/ionic-app-base/archive/master.zip
[=====] 100% 0.0s
Downloading: https://github.com/driftyco/ionic-starter-blank/archive/master.zip
[=====] 100% 0.0s
Updated the hooks directory to have execute permissions
Update Config.xml
Initializing cordova project
Adding in iOS application by default
iOS project created with cordova-ios@3.8.0
add to body class: platform-ios
Added ios platform

Your Ionic project is ready to go! Some quick tips:

* cd into your project: $ cd snapaday

* Setup this project to use Sass: ionic setup sass

* Develop in the browser with live reload: ionic serve

* Add a platform (ios or Android): ionic platform add ios [android]
Note: iOS development requires OS X currently
See the Android Platform Guide for full Android installation instructions:
```

To generate a new application called 'snapaday' using a blank template (we'll talk more about what the application will be soon). Ionic comes with some templates built in, in the example above we are using the 'blank' template, but you could also use:

```
ionic start myApp sidemenu
```

or

```
ionic start myApp tabs
```

to generate an application with a sliding menu or tabs template already set up. What's really cool though is that we can also pull in a template from codepen.io that we want to use. For example you could use the following codepen.io example:

<http://codepen.io/ionic/pen/AjgEB>

as the base for your new application by running the following command:

```
ionic start myApp http://codepen.io/ionic/pen/AjgEB
```

We're just going to stick with a boring blank template for now though. Once your application has been generated you will want to make it your current directory so we can do some more stuff to it.

> Run the following command to change to the directory of your new Ionic project

```
cd snapaday
```

If using the command prompt or terminal is new to you, you might want to read [this tutorial](#) for a little more in depth explanation.

Adding Platforms

Eventually we will be building our application with PhoneGap / Cordova, and to do that we need to add the platforms we are building for. To add the Android platform you can run the following command:

```
ionic platform add android
```

and to add the iOS platform you can run:

```
ionic platform add ios
```

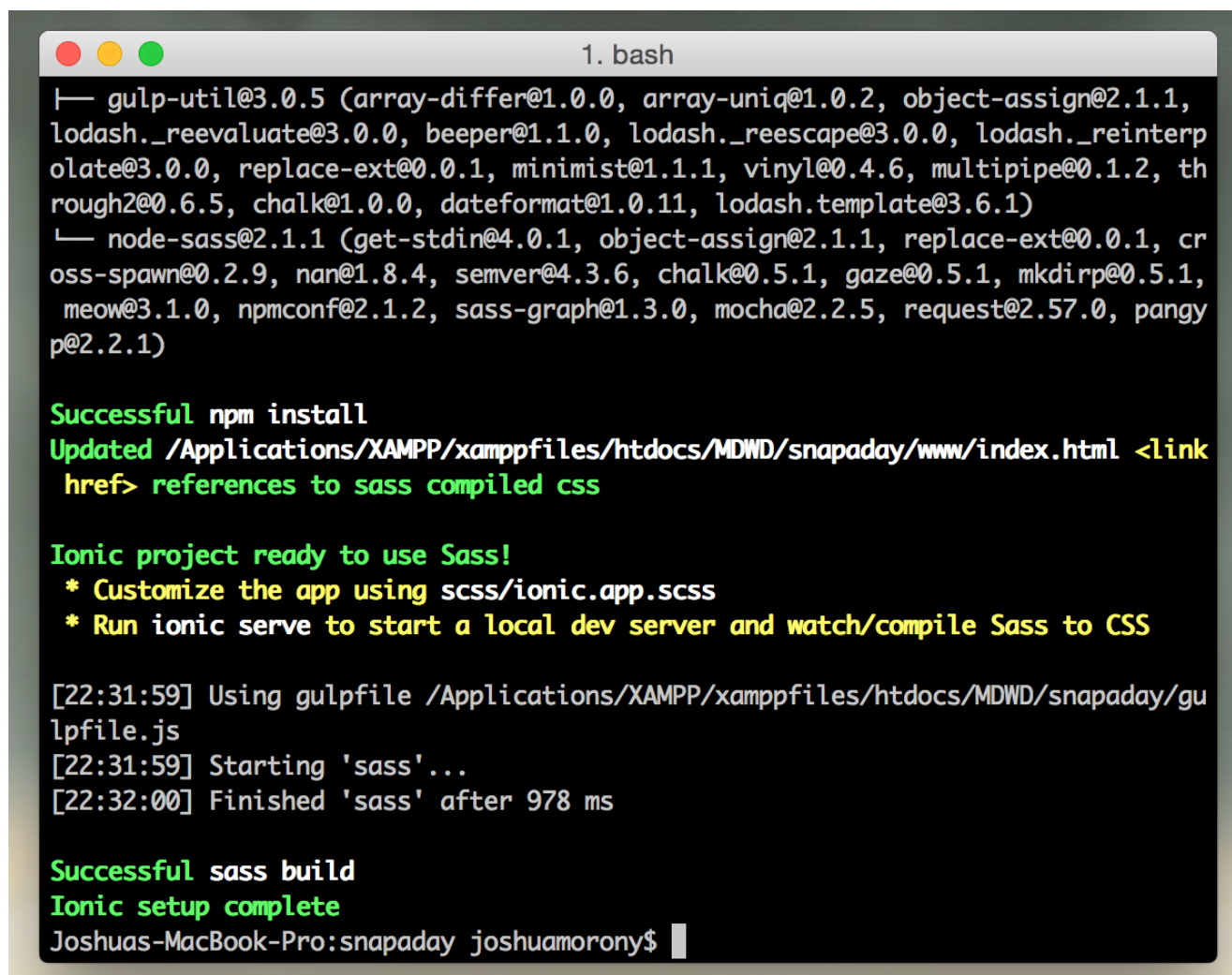
If you are building for both platforms then you should run both commands. This will set up your application so that it can be built for these platforms, but it won't really have any effect on how you build the application. As I will explain shortly, most of our coding will be done inside of the **www** folder, but you will also find another folder in your project called **platforms** - this is where all of the configuration for specific platforms live.

Setting up SASS

Ionic uses [SASS](#) (Syntactically Awesome Style Sheets) to make managing the applications CSS much easier. The general idea is that instead of editing **.css** files directly, you will edit a **.scss** file that will be compiled into a normal **.css** file. You can use a **.scss** file just like you would a normal **.css** file, but SASS also offers a bunch of extra features like **variables**, **nesting** and **mixins**. Before we can use SASS we have to set it up in the application first.

> Run the following command to set up SASS in your project:

```
ionic setup sass
```

A terminal window titled '1. bash' showing the output of the 'ionic setup sass' command. The output includes a list of installed npm packages, a 'Successful npm install' message, an 'Updated' message with a link to references to sass compiled css, a green 'Ionic project ready to use Sass!' message, two yellow bullet points for customizing the app and running the dev server, and a log of the sass build process. The terminal ends with 'Ionic setup complete' and the prompt 'Joshuas-MacBook-Pro:snapaday joshuamorony\$'.

```
1. bash
└─ gulp-util@3.0.5 (array-differ@1.0.0, array-uniq@1.0.2, object-assign@2.1.1,
lodash._reevaluate@3.0.0, beeper@1.1.0, lodash._reescape@3.0.0, lodash._reinterp
olate@3.0.0, replace-ext@0.0.1, minimist@1.1.1, vinyl@0.4.6, multipipe@0.1.2, th
rough2@0.6.5, chalk@1.0.0, dateformat@1.0.11, lodash.template@3.6.1)
└─ node-sass@2.1.1 (get-stdin@4.0.1, object-assign@2.1.1, replace-ext@0.0.1, cr
oss-spawn@0.2.9, nan@1.8.4, semver@4.3.6, chalk@0.5.1, gaze@0.5.1, mkdirp@0.5.1,
meow@3.1.0, npmconf@2.1.2, sass-graph@1.3.0, mocha@2.2.5, request@2.57.0, pangy
p@2.2.1)

Successful npm install
Updated /Applications/XAMPP/xamppfiles/htdocs/MDWD/snapaday/www/index.html <link
href> references to sass compiled css

Ionic project ready to use Sass!
* Customize the app using scss/ionic.app.scss
* Run ionic serve to start a local dev server and watch/compile Sass to CSS

[22:31:59] Using gulpfile /Applications/XAMPP/xamppfiles/htdocs/MDWD/snapaday/gu
lpfile.js
[22:31:59] Starting 'sass'...
[22:32:00] Finished 'sass' after 978 ms

Successful sass build
Ionic setup complete
Joshuas-MacBook-Pro:snapaday joshuamorony$
```

In order to use SASS you must also have gulp installed, so if that command does not run suc-
cessfully you may first have to run:

```
npm install -g gulp
```

and then once that has successfully run, retry the same command:

```
ionic setup sass
```

NOTE: If you are using a template instead of a blank application, setting up SASS can overwrite required code in the **index.html** file so make sure you check it afterwards. If you are following this tutorial exactly though you do not need to worry.

SASS is one of those things that might confuse beginners, so if you don't really *get it* then don't worry too much. Just install it and pretend you're just using a normal CSS file.

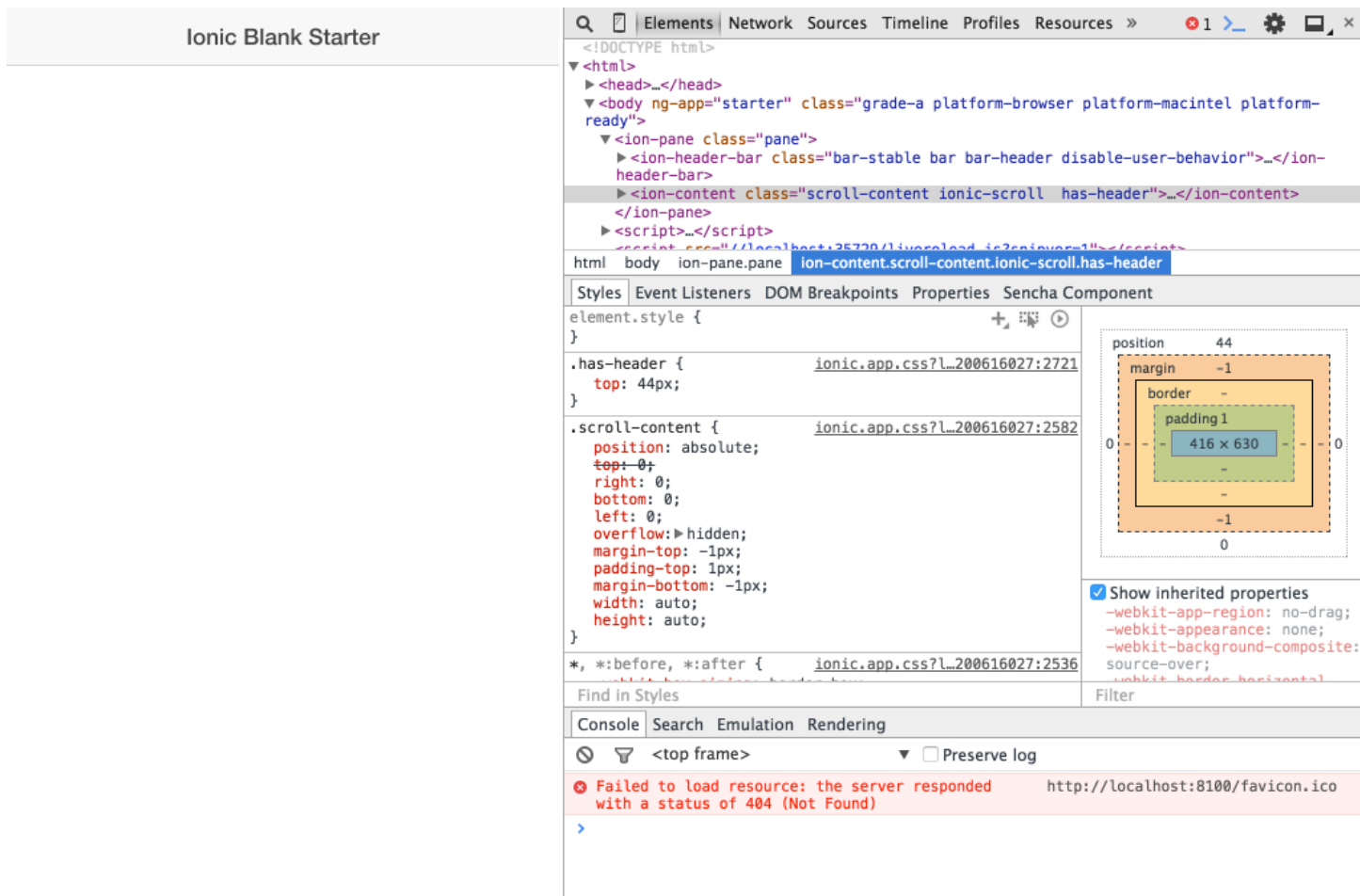
Running your application

The beauty of HTML5 mobile applications is that you can run them right in your browser whilst you are developing them. But if you try just opening up your project in a browser by going to the **index.html** files location you won't have a very good time. An Ionic project needs to run on a web server - this means you can't just run it by accessing the file directly, but it doesn't mean that you actually need to run it on a server on the Internet, you can deploy a completely self contained Ionic app to the app stores (which we will be doing). Fortunately, Ionic provides an easy way to do this.

> To view your application through the web browser run the following command:

```
ionic serve
```

This will open up a new browser with your application open in it and running on a local web server. Right now, it should look something like this:



Not only will this let you view your application but it will also update live with any code changes. If you edit and save any files, the change will be reflected in the browser without having to reload the application.

To stop this process just hit:

Control + C

when you have your command terminal open.

Upload your app to Ionic View (Optional)

You don't have to perform this step but I'd highly recommend that you do. Ionic provides an application called 'Ionic View'. You can download it from the app stores here:

iOS: <https://itunes.apple.com/au/app/ionic-view/id849930087?mt=8>

Android link: <https://play.google.com/store/apps/details?id=com.ionic.viewapp&hl=en>

This application allows you to quickly view your own applications on your device, or even send your application to others to test (which is especially handy for iOS as you don't need to worry about provisioning profiles and such - we'll talk all about those a little later).

Before you can use the Ionic View application, you will need to create an ionic.io account.

> Go to ionic.io and create an account: <http://ionic.io/>

Once you have created an account you can login through the command line.

> Login through the command line with the account you just created by running the following command:

```
ionic login
```

> Once logged in, run the following command to upload your current project to Ionic View:

```
ionic upload
```

Once that has finished uploading you can open up the Ionic View application on your device, log in, and then choose your application to view it. To update your app on Ionic View you will just need to run:

```
ionic upload
```

again, and then select your application in Ionic View and choose "Sync to Latest". This will download the latest version and allow you to view it.

The Anatomy of an Ionic Project

Ok, so now we have set up our Ionic application and have a blank template to start from. Even though we're using a blank template, there's still quite a bit of code that has been set up for us already. Now is a good time to have a bit of a poke around and see how everything is working. Take a look around in the source code of the project to see if you can get some idea of what's going on.

There's many different folders and files that make up the application, but we don't need to worry about a lot of them to begin with. A lot are files that help do behind the scenes stuff and you may never even need to touch them. In fact, most of the time when we are building the app we will only be modifying what is inside the **www** folder (i.e. the web content), everything else is mostly related to configuration for the Ionic application or for Cordova (which is included by default). So let's focus on just the contents of that **www** folder for now.

index.html

Like any web page, **index.html** is the first file that is loaded. This file even looks like any other website you may be used to. It's responsible for including all the resources we require (like the Ionic library, all of our own Javascript files and so on) and initialising the application.

An important thing to note about this file is that people using your application will never leave it. Unlike a typical website where we might navigate from index.html to about.html to contact.html, in an Ionic application we will **always** be on index.html. Many Ionic applications have (and we will too) other .html templates beside the index.html, but we will never actually load those pages. When using Ionic, we are creating a single page application that uses JavaScript to switch between different views.

js

This is perhaps the most important folder. It will contain all the Javascript you write for your application. For now it only contains a single **app.js** file which sets up the Angular module for the app and runs a little bit of code when the application is loaded and ready.

The most important thing to note about the **app.js** file is this line at the top:

```
angular.module('starter', ['ionic'])
```

What this does is create an AngularJS application called 'starter' and it includes the Ionic functionality in that app. Ionic is built on top of AngularJS, to put it simply an Ionic application is an AngularJS application with a bunch of extra features to make creating mobile applications easier.

Now if you look at the **body** tag in your **index.html** file you will see this:

```
<body ng-app="starter">
```

This attaches the Angular application that was just created to the body tag. You can attach the application to any tag you like simply by using the **ng-app** directive.

Our application isn't called "starter" though, it's called snapaday! So let's go ahead and make that small change.

> Edit your **app.js** file and make the following change:

```
angular.module('snapaday', ['ionic'])
```

> Edit your **index.html** file and make the following change:

```
<body ng-app="snapaday">
```

In future we will be adding to this file quite a bit, as well as creating additional files for other modules that we will create for our controllers and services. Most people who read this course will likely be new to AngularJS so I will explain concepts as we go, but if you want a bit of a leg up then you can read [this post](#) where I give a bit of a crash course introduction to AngularJS.

lib

This folder contains all of the Angular / Ionic libraries that our application will make use of. This is basically the engine for our application, but fortunately we'll never really have to touch anything in this folder.

templates

The templates folder will contain HTML templates that are being used throughout the application. We have one problem though, **it doesn't exist yet**. It's really important to have some structure and organisation in your application as it grows. You *could* place just about everything in your **app.js** file and your **index.html** file, but your application would quickly become

ugly and unmanageable. Very shortly we will create a templates folder and create our very own template.

css, images

These folders contain the css and image resources for our application. I mentioned setting up SASS in the previous lesson, the css folder is where the css files SASS generates will be stored. To modify the css we will be editing the **ionic.app.scss** file located in the **scss** folder (which is in the root directory of your project) but we will get to how to do this later.

As I mentioned above, there's a lot more files and folders that we haven't discussed yet. At this stage it's not important to understand what they do, as they become useful to us and we begin to make modifications here and there it'll be easier to understand rather than overwhelming you with information now.

Introducing Snapaday

We will be modifying and adding to this starter application until it becomes the app that we want to build... which is what exactly?

I wanted something that was relatively easy, touches on most of the core concepts of Ionic, and is a little more fun and original than a todo list or note taking app.

The example application that we will be building in this course will encourage users to take a photo each day, and then create a slideshow that displays all of their photos in succession, kind of like this [video](#).

Through building this application you will learn:

- The basics of Ionic, Angular and PhoneGap / Cordova
- How to build a mobile interface with lists, buttons, modals and more
- How to store, retrieve and delete data
- How to use native API's like the Camera
- How to share content to Facebook
- How to style your application with SASS and CSS

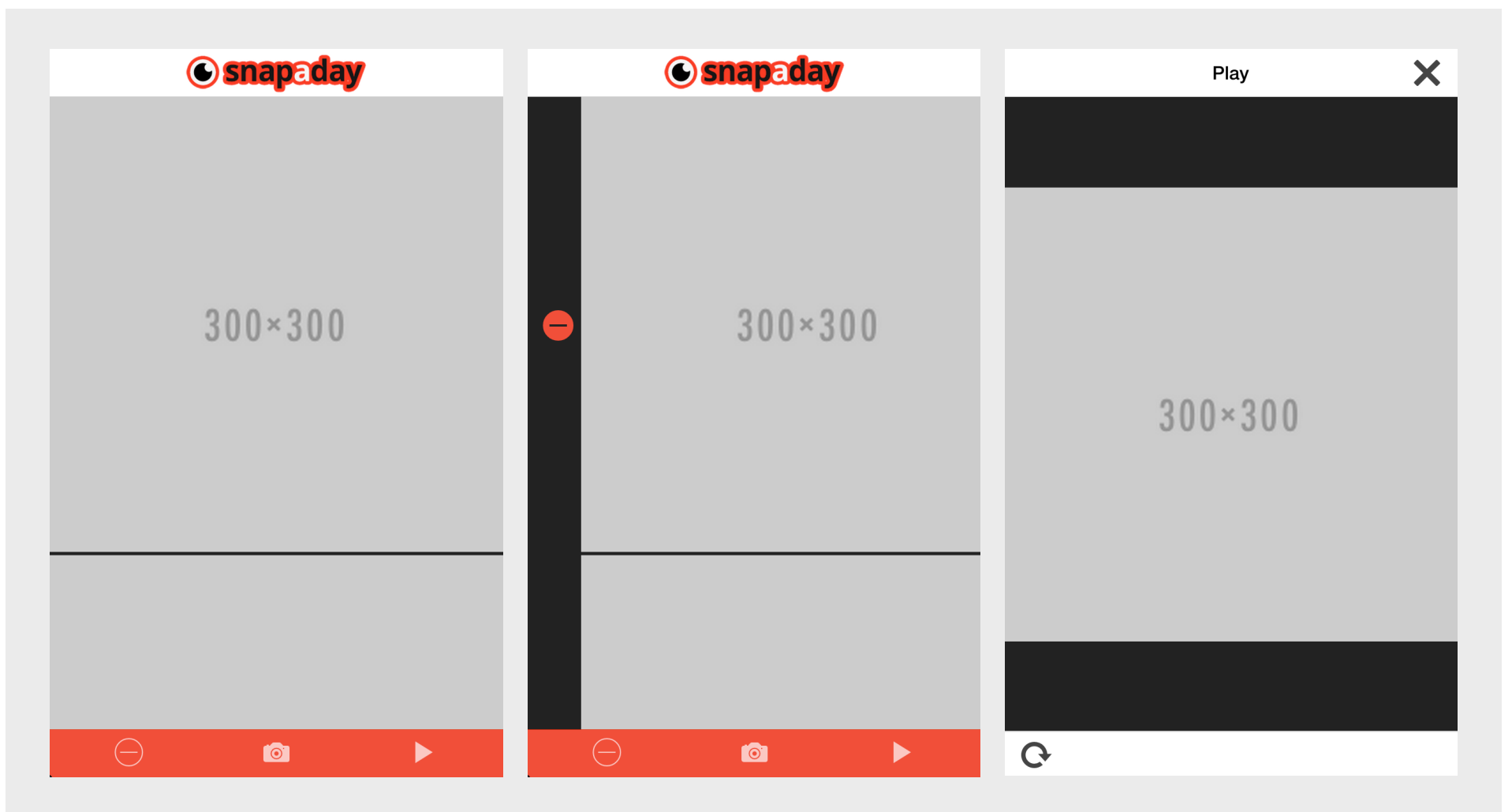
- How to minify your application and prepare it for distribution
- How to build your application with PhoneGap and PhoneGap Build
- How to sign your iOS and Android applications (this is required for submitting to the app stores)
- How to submit your application to app stores
- ...and a bunch more

The application will consist of two main views. The first view will contain a list of all the photos that have been taken, and will provide the following options:

- Delete a photo
- Take a new photo
- Play photos

If a user chooses to 'Play photos' this will launch the second screen which will display the slideshow to the user. On this screen they will have the option of replaying the slideshow or exiting from it.

Here's what the various screens will look like when it is finished:



Let's get started!

We're going to take some baby steps to start off with by making sure we create a nice, scalable structure for our application. In the next lesson we'll get into some more serious stuff, but there's nothing more important than having a strong foundation to build on top of.

There's a few different ways to organise an Ionic application, technically you can do it anyway you like that works, but we will be implementing one approach that is reasonably popular and simple.




















NOTE: *We will be building the entire application step by step. If you follow each of the instructions that are:*

> highlighted like this

then you should end up with the final product. There's a lot of steps though and chances are something will go wrong at some point. In your download package you will have received the source code for this application, split into various lessons. If you ever get stuck or something isn't working quite right, just compare your code to the sample code in "snapaday-lesson-0X".

To start off with we're going to create a **templates** folder that will hold the HTML templates for all of the different views within our application.

> Create a new folder inside of **www** called **templates**

Name	^	Date Modified	Size
 .DS_Store		Today 11:05 pm	6 KB
 .gitignore		Today 10:22 pm	138 bytes
 bower.json		Today 10:22 pm	118 bytes
 config.xml		Today 10:25 pm	860 bytes
 gulpfile.js		Today 10:22 pm	1 KB
▶  hooks		Today 10:22 pm	--
 ionic.project		Today 10:31 pm	155 bytes
▶  node_modules		Today 10:29 pm	--
 package.json		Today 10:25 pm	577 bytes
▶  platforms		Today 10:25 pm	--
▶  plugins		Today 10:25 pm	--
▶  scss		Today 10:22 pm	--
▼  www		Today 11:05 pm	--
▶  css		Today 10:32 pm	--
▶  img		Today 10:23 pm	--
 index.html		Today 10:31 pm	783 bytes
▶  js		Today 10:22 pm	--
▶  lib		Today 10:22 pm	--
▶  templates		Today 11:05 pm	--

> Modify the code in your **index.html** file to look like the following:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="initial-scale=1, maximum-scale=1, user-
scalable=no, width=device-width">
    <title></title>

    <!-- compiled css output -->
    <link href="css/ionic.app.css" rel="stylesheet">

    <!-- ionic/angularjs js -->
    <script src="lib/ionic/js/ionic.bundle.js"></script>

    <!-- cordova script (this will be a 404 during development) -->
    <script src="cordova.js"></script>
```



```

    <!-- your app's js -->
    <script src="js/app.js"></script>
</head>
<body ng-app="snapaday">

    <ion-nav-bar class="bar-light">
        <ion-nav-back-button>
        </ion-nav-back-button>
    </ion-nav-bar>

    <ion-nav-view></ion-nav-view>

</body>
</html>

```

We've made a little change to the header here, but the most important thing to note is the inclusion of `<ion-nav-view>`. The easiest way to explain this is to imagine it as a placeholder for our templates. We could create a template called **template1.html** and trigger it to be displayed with this `<ion-nav-view>` at some point, and then we could have another **template2.html** which we could switch to and then that would be displayed instead. Let's create our first template.

> Create a new file called **main.html** inside of the **templates** folder and add the following code to it:

```

<ion-view hide-back-button="true" view-title="Main View">
    <ion-content>
        I'll do something cool soon enough!
    </ion-content>
</ion-view>

```

To create templates that will be displayed inside of we use `<ion-view>`. By supplying a **view-title** we can dynamically change the header in **index.html** and we can also control whether or not we want to display a back button so that the user can navigate back to the view that they came from previously.

If you take a look at your application by running `ionic serve` now you will see:

Oops... now we can't see anything. We've set up the and a template to be displayed inside of it, but how is our application supposed to know what we want to display? We can do this using **routes**.

> Modify your **app.js** file to look like the following:

```
angular.module('snapaday', ['ionic'])

.run(function($ionicPlatform) {
  $ionicPlatform.ready(function() {
    // Hide the accessory bar by default (remove this to show the accessory bar
above the keyboard
    // for form inputs)
```

```

        if(window.cordova && window.cordova.plugins.Keyboard) {
            cordova.plugins.Keyboard.hideKeyboardAccessoryBar(true);
        }
        if(window.StatusBar) {
            StatusBar.styleDefault();
        }
    });
})

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

    $stateProvider
    .state('main', {
        url: '/main',
        templateUrl: 'templates/main.html',
        controller: 'MainCtrl'
    });

    $urlRouterProvider.otherwise("/main");

});

```

You will notice that we have added a **config** that is injecting the `$stateProvider` and `$urlProvider` services. These services are provided by default and can help us handle routes within our application (a route is essentially a way to map URLs to views). By "injecting" these services (which means we have included them as parameters) we make the service available within the scope that it is being called from. I don't want to get too technical just yet so don't worry if you don't really get what all this means, we will continue injecting and using services throughout our application and it will become easier to see how it all works.

In the code above we are creating a **state** called **main**. A **state** is the state that your application is in, for example: you might have a main state, an introduction state, and a login state within your application. Each of these states can have different **templates** attached to them, which we specify through the **templateUrl**.

The **url** is used to determine when this state should be active, in this case it has a URL of **/main** so if our application is currently pointing to:

localhost:8100/#/main

Then the main state will be shown. This allows us to link (similar to how you would on a normal website) to different states very easily. If the current route does not match anything we have listed then it will default to **/main** since we have specified:

```
$urlRouterProvider.otherwise("/main");
```

Finally, we've supplied a **controller**. A controller can be considered somewhat like a director or traffic controller in your application. For example, if you want something to happen when somebody taps a button, then you would create a function in your controller to do this and you would call it from one of your templates. By specifying a controller for this state, we are saying that we want this controller associated with this template. So if we create a function called `doSomething()` in our Main controller then we can call that function from our **main.html** template. But if we were to create another template with a different controller then we could not call that function (you can however use the same controller for multiple templates).

If we try to view our application now we're still going to be running into problems. We've assigned a controller to our state but it doesn't exist yet, so now we're going to have to create that.

Again, we *could* place this controller in the **app.js** file but we want to keep our code organised. So we're going to create an additional two files: **controllers.js** to hold all of our controllers and **services.js** to hold all of our services. We don't have any services just yet, but we do want to create our Main controller, so let's do that.

> Create a new file at **www/js/controllers.js** and add the following code:

```
//This creates a new module called 'controllers' that will hold all of the controllers for our app  
angular.module('controllers', [])  
  
.controller('MainCtrl', function($scope, $state) {  
  
    console.log('MainCtrl');  
  
});
```

In the code above we've created our Main controller (which doesn't do a whole lot just yet). Notice that in **controllers.js** we are creating a new **angular.module** just like we have in **app.js**. An

Ionic app is made up of lots of modules: the app itself is a module, controllers are modules, services are modules and so on. Modules are great because it allows us to separate code which makes it organised and reusable. Although we don't have a service we want to create yet, let's get our **services.js** file set up anyway.

> Create a new file at **www/js/services.js** and add the following code:

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

We also need to include our new controllers and services modules in the apps main module. To do this we supply the controllers module and the services module as a requirement for the main module.

> Add the controllers and services modules as a requirement in the apps main module in **app.js**:

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

And finally, we also need to include these new files we've created in our **index.html** file.

> Add the following references to **index.html**:

```
<script src="js/app.js"></script>
<!-- Make sure to include any new javascript files you create -->
<script src="js/controllers.js"></script>
<script src="js/services.js"></script>
```

Run `ionic serve` to test your application in the browser (you can have this running constantly in the background) and make sure you see the following:

Main View

I'll do something cool soon enough!

If something has gone wrong then try taking a look around the source code to see if you can find out what's wrong. Make sure to check the 'Console' in your debugging tools as a first step (Right Click > Inspect Element > Console), as often it will tell you where the error is. If you can not figure out what's wrong, then compare your code against the code for Lesson 1 that was supplied with this course. We will discuss some more ways to debug your applications in a later lesson.

We have a reasonably scaleable and easy to understand application structure now. Even this structure will become unmanageable if the application is too large or complex though. You could further break down the controllers module into individual controllers (rather than having

all of your controllers in one file) or use one of the methods discussed in the homework tasks below. As a beginner though, this is a great structure to start off with.

In the next lesson we will start getting our hands dirty and creating a lot more code of our own by building out the views for our application. We've made a pretty good start already in the first lesson, so far we have:

- Installed Ionic and Cordova
- Generated a new application
- Setup SASS
- Run the application through a browser
- Run the application on a device with Ionic View
- Created a template
- Created a controller
- Set up routing

This gives us a really nice base to work off of moving forward. At this point we have a pretty generic starter application with our development environment set up and ready to go. You would repeat these steps for just about every new Ionic app that you want to create (except for the things we had to install of course).

As we build the application and come across new concepts, I will explain everything as it comes up. So don't worry if you don't have any idea what you're doing yet (that's pretty much the idea anyway!).

Homework Tasks

- Find a template on codepen.io that you like and generate a separate application with it (make sure you don't generate it inside of the directory of the app you generated in this lesson!)
- Spend 10 minutes looking through the different files that have been generated, no need to understand them just have a poke around and get a bit of a feel for the application.
- Read my [Angular + Ionic crash course](#) article

SECTION 4

MOBILE GAME DEVELOPMENT

Lesson 1

GETTING STARTED WITH PHASER

If you've read through the complete version of this course then you would've learned how to build a simple data driven application with [Ionic](#) by now. Typical business style applications are where HTML5 mobile frameworks like Ionic excel. They can't do everything though. If you want to build a game then you are going to have to use a different technology. Ionic does not provide functions that would be necessary for typical games, and the way in which the framework is designed and runs is just not suitable for those sorts of applications (Ionic certainly doesn't come with an in built physics engine, Phaser on the other hand has three!).

Fortunately, there are still options for games in the HTML5 mobile space. We're going to be looking at using the [Phaser](#) framework to build a simple mobile game. It's important to note that Phaser is not specifically a framework for mobile, many developers are creating browser based HTML5 games to run on the desktop.

Like Ionic, since Phaser uses HTML5 technology then it can run on mobiles and be distributed through app stores just like native apps. When using HTML5 to build mobile games you do

need to keep the performance limitations of mobile browsers in mind - HTML5 isn't a great option if you want to build 3D games with HD graphics and explosions, but if you want to build reasonably simple games (think Flappy Bird, Puzzle games, Platformers etc.) then HTML5 is a great choice.

Phaser is currently one of the most popular HTML5 game frameworks so I doubt you will regret the decision to pursue it, it is by far the most active framework on html5gamedevs.com and is being actively developed. Before we jump into building our first game, let's explore the environment a little and take a look at some examples that can be built with Phaser.

Pre-lesson Tasks

- Take a look around the [Phaser website](http://phaser.io). Check out the [documentation](#) and [examples](#)
- Take a look at some of Thomas Palef's games from his [12 games in 12 weeks challenge](#) (he built these as a complete beginner)
- Take a look at a slightly more complex "Angry Birds" style game build with Phaser: [Sasquash](#)
- Read through Phasers [Making Your First Phaser Game](#) guide. There's no need to actually code the app yourself just yet, just take a look and see if you can make sense of it.

Getting Started with Phaser

Install XAMPP

Just like Ionic, Phaser needs to run on a webserver on your machine. Ionic has a nice little command built in to do that for you, but with Phaser you will need to set it up yourself. What you want is to be able to store all of your files locally and access them through the **http** protocol like this:

<http://localhost/MyCoolGame>

instead of through the file system like this:

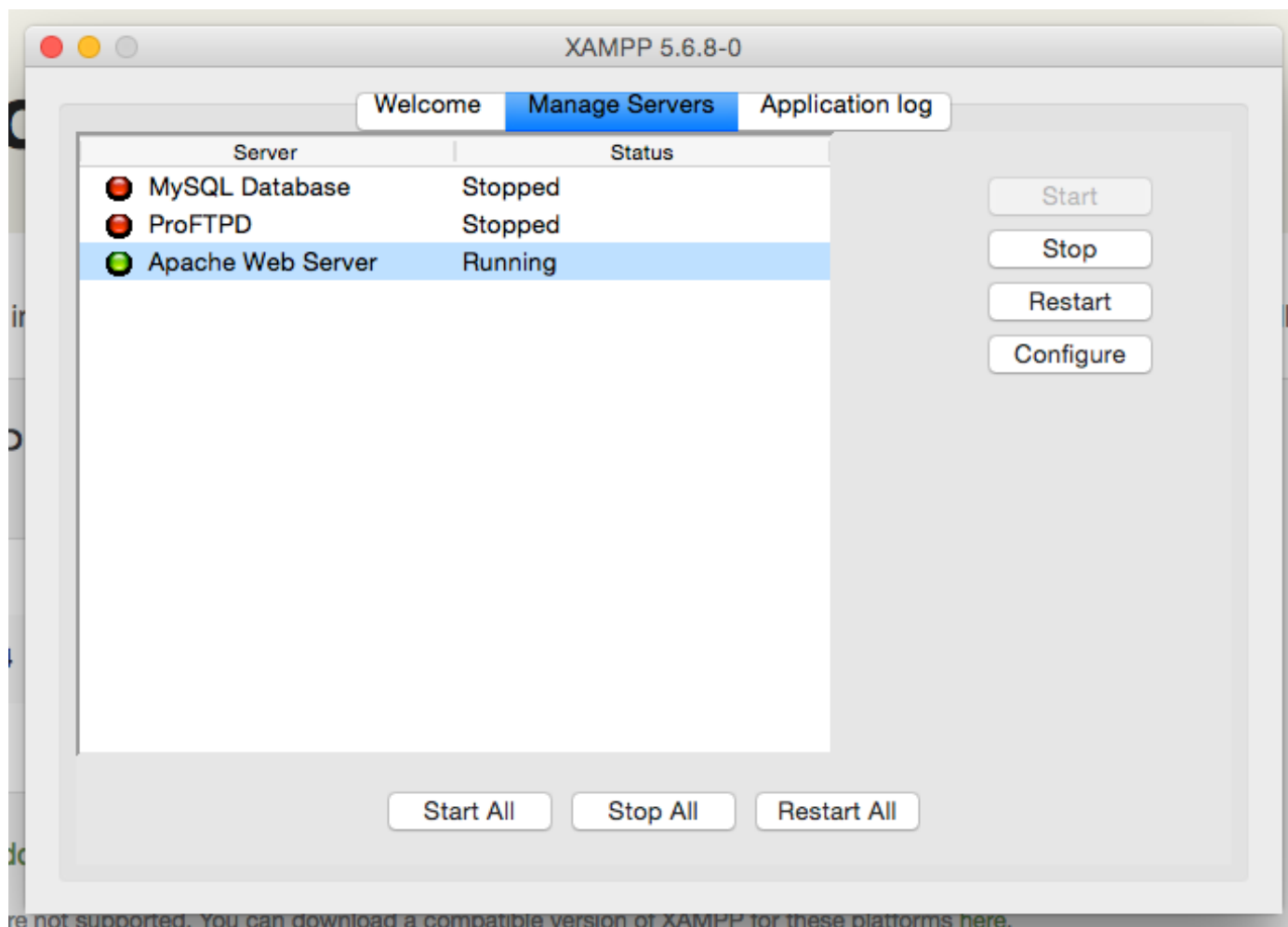
`file:///Users/blah/blah/MyGame`

because that doesn't work. I won't get into the technicalities behind *why* that doesn't work, but it has to do with the file:// protocol being much more restricted.

There are other methods to getting a webserver running on your machine, so feel free to do whatever you like, but a great option for both Mac and PC is [XAMPP](#).

> [Download](#) and Install XAMPP

Once you have XAMPP set up you should be able to access the XAMPP control panel:



> Open the XAMPP control panel and make sure the Apache Web Server is running

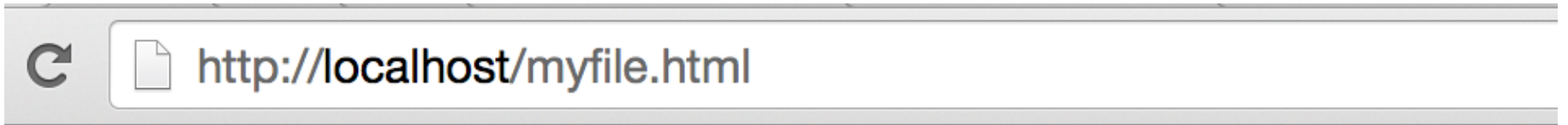
Now any files you store in your htdocs directory you will be able to access through localhost in the browser. On a Windows machine this will be located at:

c:\xampp\htdocs\

and on a Mac:

/Applications/XAMPP/htdocs

So if you store a file in `htdocs/myfile.html` you will be able to access it by going to `http://localhost/myfile.html` in the browser:



Download Phaser

Phaser is extremely easy to set up, it is literally just a single javascript file that you need to include. There's a few different ways to grab the source file, and you can even create a custom build that only includes the bits you need (which keeps the file size down), but for now we will just be downloading the full `phaser.min.js` file.

> Go to the [download page](#) and download the **min.js** version of Phaser

We will go through setting this up to be used in an actual game in just a moment, but first let's talk a little bit about the game we will be building with Phaser.

Introducing: I Hate Squares

After completing the tasks above you should be ready to get started with Phaser, and you should at least be somewhat familiar with what Phaser is all about.

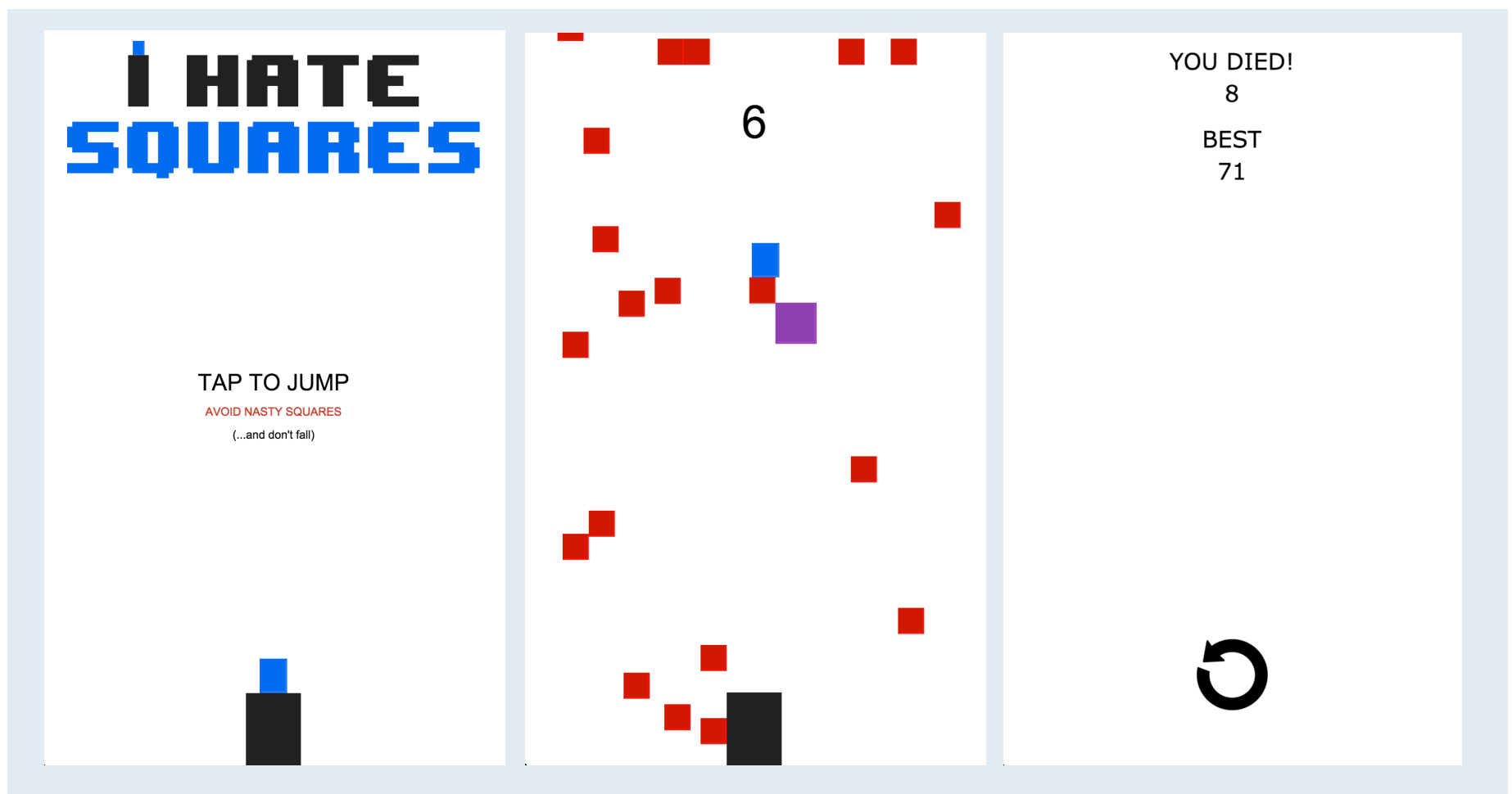
Now we need to figure out what we actually want to build. I wanted to avoid building something cliché like a Space Invaders or Snake clone and I also want it to be relatively simple. I like building simple, difficult and addictive games for mobile - they're reasonably quick to build, small enough for one developer to handle and can be very successful on the app stores - so in that fashion I came up with "I Hate Squares".

The premise is simple, you're a rectangle in a world full of squares who want to kill you. The game will work as follows:

- The player will be a blue rectangle

- The player will be trapped within an arena (which will expand to the size of whatever device it is running on)
- The player will be able to move within the arena
- Enemies (red blocks) will randomly spawn and enter the arena from the left or right
- If the player touches an enemy, or falls out the bottom of the arena, it will be game over
- The score will be tied to how long the player can survive

The final product will look something like this:



We are developing for mobile so we must consider opportunities and challenges that are unique to mobile when designing our game. Here's a few things to think about:

- The game will be played on different devices (phones and tablets) with different screen sizes and resolutions
- The primary input method will be touch
- Consider how the game will be held; will your users hand block an important part of the interface? Will parts of the interface be hard to reach with the thumb?
- Processing power will be smaller than that of a desktop computer, and some devices will be more powerful than others

- We have access to features that a desktop computer doesn't: motion and GPS for example

Applying these considerations to our game, the most prominent issue that comes to mind is movement. Our character will be able to move in any direction - this would be easy to achieve with the use of the arrows on a keyboard but we don't have access to that. So our options may include:

- Control the character by dragging with touch
- Create our own up / down / left / right controls on the screen
- Use the devices accelerometer to control the movement by tilting the device
- Control the character with touch in some other way, perhaps swiping to gain momentum

We will also need to deal with the changing size of devices and how that will affect our game. Fortunately this should be reasonably easy to cope with since our game space does not have to be any specific size.

We will discuss and deal with these issues as we get to them, but it's good to think of these things and consider potential approaches from the start.

Let's Get Started

Ok, enough talking - let's get our hands dirty. In this first lesson we're going to set up a skeleton for our Phaser game. We will be building a generic skeleton that can be used for just about any game.

The game in the [Making Your First Phaser Game](#) guide only consisted of a single main **state**, we will be adding an additional 4 states giving us a total of 5. A **state** in Phaser is basically a JavaScript object that runs a specific **state** that the game is in. For example one **state** might be the games title screen, another would be the game itself, and another would be the game over screen. We also might have different states for different levels, e.g: Level 1, Level 2, Level 3 etc. But we also have states for other things like booting and loading the game. The states we will be creating for our skeleton application are:

Boot

This is our first state, and it will be invoked as soon as our game is launched. We won't be using this state for a lot, just to initially scale the dimensions of the game to the appropriate size and then call our second state.

Preload

Preload is our second state and will be used to load in all the assets (images, audio, etc.) our game needs to function. The next state won't be triggered until all the assets are ready.

GameTitle

This state will be used to display the games title screen. Usually we will display the title of the game here, maybe some graphics and a 'Play' or 'Start' button to take the player into the game. Perhaps we might even want to add an 'Instructions' button in here too that could open up it's own Instructions or Tutorial state.

Main

Finally we get into the fun part. The main state will contain our actual game code.

Game Over

Eventually our player is going to die or win, at which point they will arrive in this state. We will use this screen to display a score and offer them the opportunity to start a new game.

Any good code base is modularised in some way. In web development we separate our HTML, CSS and JavaScript into separate files - this makes our code more readable and organised, and makes it easier to update in the future. In Phaser, rather than having all the code in one giant 'main' state we separate it out into different states.

Setting up the application

By now you should already have a web server like XAMPP set up on your computer and you should also have downloaded the **phaser.min.js** source file. We will start by creating a new a new folder called `ihatesquares` in the web server root, which would look something like:

`C:\xampp\htdocs\ihatesquares`

or

`/Applications/XAMPP/htdocs/ihatesquares`

> Create a folder called `I Hate Squares` inside of your `htdocs` folder

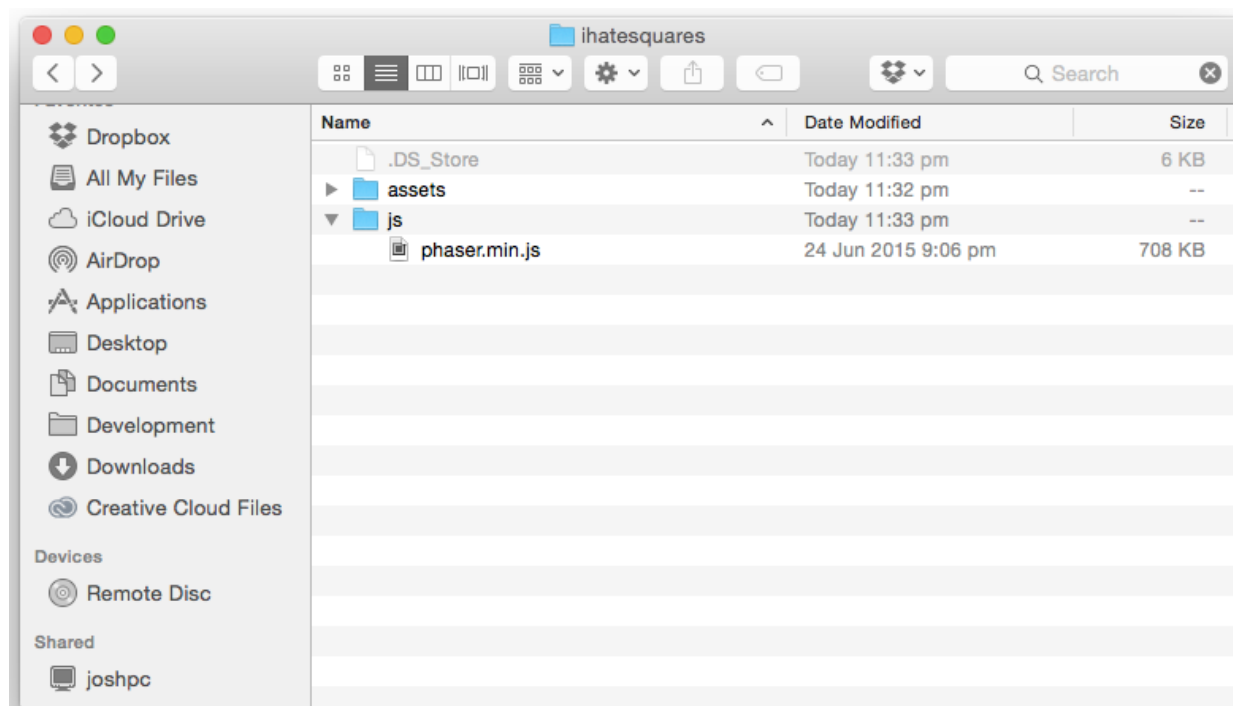
Now inside of that folder we're going to create two more folders:

> Create an `assets` and `js` folder inside of your `ihatesquares` folder

Once you've created those folders, you will need to include the **phaser.min.js** file in your project:

> Copy **phaser.min.js** into the `js` folder you just created

Your project structure should look like this:



Now we will get onto creating each of the states we just mentioned.

Creating the Boot State

> Create a file called **boot.js** inside of your `js` folder and add the following code to it:

```
var Boot = function(game){  
  
};  
  
Boot.prototype = {  
  
  preload: function(){  
  
  },  
  
  create: function(){  
    this.scale.scaleMode = Phaser.ScaleManager.SHOW_ALL;  
    this.scale.setScreenSize(true);  
    this.game.state.start("Preload");  
  }  
}
```

Currently the only role this state has is to set the screen size to fit the device that we are on, and then trigger our "Preload" state. This state will run when the game is first started and will only run once.

Detour!

Although this course is not intended to be an introduction to the JavaScript language, I feel it would be worth explaining the structure used in this code example. This is the basis for all of our states so it's important to understand what is going on. I would recommend reading through this but if you *really* just want to keep building the game then feel free to skip over it and come back later.

NOTE: *If you are new to JavaScript you can still probably make your way through this course without too much difficulty (assuming you have some programming knowledge). It'll definitely help to get the basics of JavaScript down though - try [this](#) to get started.*

In JavaScript a function acts like an object, and it can contain methods (functions) and properties. An important property of these function objects is prototype. prototype is a property that is available whenever you create a new function, however it will initially be an empty object. We can add methods and properties to this empty object which can be accessed through the object.

In the example above, we are overwriting the entire prototype object and supplying two methods. This would be similar to doing the following (without overwriting):

```
Boot.prototype.preload = function(){  
  
}  
  
Boot.prototype.create = function(){  
  
}
```

We will never have to do this in Phaser, but for the sake of an example we could call the preload method of boot with the following code since we have it defined in the prototype:

```
Boot.preload();
```

Ok, I know this is dragging on a little bit and you probably want to get into making games - just bare with me a little longer. Now we could also achieve a similar effect by creating an object like this:

```
var Boot = function(game){  
  
    this.preload = function(){  
  
    }  
  
    this.create = function(){  
  
    }  
  
}
```



```
}
```

Now when we call `boot.preload()` it will first look for a function called 'preload' defined within the object as we have above. If it can not find it within that object it will look at the objects prototype for the function (where we originally defined it).

So what's the difference? Can't we just pick one method and move on with our lives?

There's two main reasons we are using the prototype method:

1. If we create an instance of an object and try to call one of its methods, as I mentioned before it will first look within that instance and then it will look at the prototype. Since all instances will share the same constructor and thus the same prototype, if we modify one of the methods through the prototype:

```
Boot.prototype.preload = function(){  
    console.log("we changed something!");  
}
```

Then this method will effect all methods that have already been created. If we were to define the methods directly on the object then we would need to modify the method for each instance that has already been created.

2. It is also better performance wise. Methods defined directly on the object will be created every time a new instance of that object is created, whereas methods defined on the prototype are created once and then inherited by each instance.

All of the above was not at all essential for you to know to learn Phaser. If you don't really get it (I know I didn't for a long time, and I still probably don't) then don't worry too much, I just want to explain why we are doing things that may seem a little out of the ordinary rather than asking you to take them at face value. As Elon Musk stated on learning recently:

"It is important to view knowledge as sort of a semantic tree -- make sure you understand the fundamental principles, ie the trunk and big branches, before you get into the leaves/details or there is nothing for them to hang on to." - Elon Musk

Back to game making!

Seriously now, we're going to get back into the game making:

Create the Preload State

Create a file called **preload.js** and save it in your js folder, then add the following code to it:

```
var Preload = function(game){};

Preload.prototype = {

    preload: function(){

    },

    create: function(){
        this.game.state.start("GameTitle");
    }
}
```

Eventually we will be adding some more code to the preload method of the Preload state, which will load in all the assets we require for our game. Like our Boot state, Preload also has a create method that will run once when the state is created. In this case all we are doing is using it to call our next state.

Create the GameTitle State

> Create a file called **gametitle.js** and save it in your js folder, then add the following code:

```
var GameTitle = function(game){};

GameTitle.prototype = {

    create: function(){

    },

    startGame: function(){
        this.game.state.start("Main");
    }
}
```

```
}
```

Again, GameTitle also has a create method that will run when it is created. We will use this to create our game title screen which will contain some graphics and some form of 'Start' button that will be used to start the game when the player is ready. In this case we don't call our next state, Main, until we receive input from the user. So we will trigger the 'startGame' method from somewhere within the create method (in most cases we would trigger the startGame() function when a user taps on a button).

Create the Main State

> Create a file called **main.js** and save it in your js folder, then add the following code:

```
var Main = function(game){  
  
};  
  
Main.prototype = {  
  
  create: function() {  
  
  },  
  
  update: function() {  
  
  },  
  
  gameOver: function(){  
    this.game.state.start('GameOver');  
  },  
  
};
```

This is the state where most of the magic happens, and it is responsible for running the actual game itself. We have the usual create method as we did in our other states, and a gameOver method that we will use to trigger the next state when we are ready, but there is one method here we haven't added to the others. The update method will constantly loop every frame refresh. We will create the initial state of our game in the create method, but will place a lot of the logic for our game in the update method.

The update method may run every, let's say for example, 100 milliseconds - so 10 times every second. Inside of here we might test for things like:

"Is the player currently colliding with an enemy?"

"Is the user pressing the space bar button?"

"Is the player currently moving?"

Then we can act upon what is currently happening in the game. If the player is colliding with the enemy then we may want to trigger the `gameOver` method, or maybe we want to instead take away one life from the player. If the player is holding down the space bar button maybe we want to make the player fly until they let go, and so on.

Create the GameOver State

> Create a file called **gameover.js** and save it in your `js` folder, then add the following code:

```
var GameOver = function(game){};

GameOver.prototype = {

  create: function(){

  },

  restartGame: function(){
    this.game.state.start("GameTitle");
  }

}
```

Finally we've arrived at our final state, and there's no surprises here. We simply have our usual `create` method which will be used to create the Game Over screen, as well as an additional method that will allow the user to restart the game. When we leave a state and come back to it again later, it will be completely refreshed. So when we eventually call:

```
this.game.state.start("Main");
```

again, our create method in the main state will run again, creating a fresh new game. It is possible to create some persistence throughout games (perhaps giving the user the ability to collect coins which they can spend) through using local storage, but we will talk about that later.

Bringing it all together with index.html

As is the case with a normal website, the first file that is loaded is our **index.html** file. We will use this file to initialise our Phaser game, load in our javascript files (our states) and to call our initial state.

> Create a file called **index.html** and place it in the root folder of your project. Then add the following code:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>I Hate Squares</title>
  <script type="text/javascript" src="js/phaser.min.js"></script>
  <script type="text/javascript" src="js/boot.js"></script>
  <script type="text/javascript" src="js/preload.js"></script>
  <script type="text/javascript" src="js/gametitle.js"></script>
  <script type="text/javascript" src="js/main.js"></script>
  <script type="text/javascript" src="js/gameover.js"></script>
  <style type="text/css">
    body {
      margin: 0;
    }
  </style>
  <script type="text/javascript">
    (function() {

      //Create a new game that fills the screen
      game = new Phaser.Game(window.innerWidth *
window.devicePixelRatio, window.innerHeight * window.devicePixelRatio,
Phaser.AUTO);

      //Add all states
      game.state.add("Boot", Boot);
      game.state.add("Preload", Preload);
      game.state.add("GameTitle", GameTitle);
```

```

        game.state.add("Main", Main);
        game.state.add("GameOver", GameOver);

        //Start the first state
        game.state.start("Boot");

    }());
</script>
</head>
<body>
</body>
</html>

```

The embedded JavaScript code above is responsible for initialising our game (this will run as soon as the page is loaded). To create an instance of a Phaser game we call `Phaser.Game()` and supply it with the width and height of our game, as well as which rendering engine we want to use.

We don't just supply a width and height though, we also get the `devicePixelRatio` of the device we are running on. This is not always necessary but since we are creating games for a mobile device we do this so the game scales properly. We can get the width and height of the space we have to work with by using `window.innerWidth` and `window.innerHeight` (this will make the game full screen). The reason we multiply it by the pixel ratio though is because different devices can have different pixel densities. Take Apple's Retina displays for example, if we were to put this game on an iPhone 5 and tried to use `window.innerWidth` and `window.innerHeight` we would get a size of:

320 x 568

But the actual size of the screen is 640 x 1136. This is because the iPhone 5 has a device pixel ratio of 2, so if we multiply the `innerWidth` and `innerHeight` by 2 we get the actual screen size of:

640 x 1136.

As I mentioned, we also supply which rendering engine we want to use to the `Phaser.Game()` call. This can either be `WebGL` or `Canvas`, if we set it to `AUTO` then it will use `WebGL` if available, but fall back to `Canvas` if it is not available.











Once we have initialised our game, we add all of our states to the game:

```
game.state.add("Boot", Boot);
game.state.add("Preload", Preload);
game.state.add("GameTitle", GameTitle);
game.state.add("Main", Main);
game.state.add("GameOver", GameOver);
and trigger our first state with the following code:
```

```
game.state.start("Boot");
```

This will then start the chain reaction of calling the states we created before: Boot calls Preload which calls GameTitle which calls Main which calls GameOver which then calls GameTitle once more.

This brings us to the end of this lesson. Now we have a blank skeleton application that you could use for just about any game. Your project structure should look something like this:

Name	^	Date Modified	Size
 .DS_Store		Today 12:15 am	6 KB
 assets		Yesterday 11:32 pm	--
 index.html		Today 12:03 am	1 KB
 js		Yesterday 11:58 pm	--
 boot.js		Yesterday 11:36 pm	235 bytes
 gameover.js		Yesterday 11:59 pm	157 bytes
 gametitle.js		Yesterday 11:50 pm	149 bytes
 main.js		Yesterday 11:54 pm	175 bytes
 phaser.min.js		24 Jun 2015 9:06 pm	708 KB
 preload.js		Yesterday 11:47 pm	148 bytes

Remember that all of the code for each lesson is available in the download package you would have received. So if you have any trouble throughout this course make sure to check your code against the source code for the lesson you're up to. I recommend that you don't just copy and paste though and that you actually code it yourself, you'll learn a lot more this way.

In the next lesson we will get into actually building our own game!

Homework Tasks

- If you haven't already, complete the 'Pre-lesson Tasks' earlier in this lesson

SECTION 5

APP STORE SUBMISSION

Lesson 1

NOT AVAILABLE IN PREVIEW

NOTE: This is a preview version of Mobile Development for Web Developers. It includes the first lessons from both Section 3 and Section 4. If you would like to purchase the entire course, please visit: <https://joshmorony.com/mobile-development-for-web-developers/>