Learn Angular

BY MAKING A QUIZ APP FROM SCRATCH

HUNGRYTURTLECODE.COM

Course Index

Chapter 1

I've Learnt Some JavaScript! What Now?

That may be something that pretty much all of us have said at some point or another. You have learnt some basic programming syntax, but you now want to build something. In comes this AngularJS Project.

A great way to continue your learning with JavaScript is to learn a framework. So in this course, we will be doing just that. The framework of choice this time is AngularJS.

AngularJS is a framework that has been built by Google and is very widely used for web projects for small companies and huge ones alike.

Learn AngularJS by doing

By far the best way to learn anything is to just dive in and do it. So that is exactly what I am going to urge you to do.

This article and the below video are part one of a whole course that has been designed with beginners in mind. So it does not include a lot of heavy theory and ideas.

Instead it tries to give you the minimum effective dose for Angular. Taking your existing knowledge of Javascript and building on top of that with some new Angular code.

This results in some of the code not being perfect “Angular”. So if you have used Angular before you may want to tell me off, but it is done like this intentionally to help those beginners.

What Will We Be Building in This AngularJS Project?

Take a look at the video below (which is also a video version of this written tutorial) and you will see the application in action.

Or you can see the finished working application here.

The application itself is a quiz application with a little learning “Facts” area for users to swot up before they take the quiz. The user will then get their results when they are finished.

Check out the git repo for this project

Let’s Get Going!

Click the image to go to video on youtube, or go to https://www.youtube.com/watch?v=yordL7Yczes&index=1&list=PLqr0oBkln1FBmOjK24_B4y_VAA8736wPq

So for those of you that prefer to read the tutorial. Let’s get started.

The Code To Get Started With

This is the HTML markup that we will need to get going with the project.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Turtle Facts and Quiz</title>
    <!-- Bootstrap css and my own css -->
    <link rel="stylesheet"
        href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css"
        integrity="sha384-1q8mTJOASx8j1Au+a5WDVnPi2lkFfwwEAa8hDDdjZlpLegxhjVME1fgjWPGmkzs7"
        crossorigin="anonymous">
    <link rel="stylesheet" href="css/style.css">
</head>
<body>

    <div class="container">
        <div class="page-header">
            <h1>Turtle Facts Quiz</h1>
            <h3>
                Learn about all the turtles below before you decide to take on the
                <strong>TURTLE QUIZ</strong>
            </h3>
        </div>

    </div>

    <!-- third party js -->
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.5.0-rc.2/angular.min.js"></script>
    <script src="https://code.jquery.com/jquery-2.2.0.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.min.js" integrity="sha384-
    0mSbJDEHialfmuBBQP6A4Qrprq5OVfW37PRR3j5ELqxss1yVqOtnepnHVP9aJ7xS" crossorigin="anonymous"></script>

    <!-- Our application scripts -->

</body>
</html>
```

It is simply a bootstrap layout with jQuery and Angular added in. I have added some basic markup at the top of the container which will be present throughout the whole application.

Let Yourself Watch Your Code LIVE!

If you have been writing HTML for any length of time you will know how annoying it can be to write some code, then have to go into the browser to refresh.

So before we get really stuck into this project I am going to make a suggestion. Live-server.

What is Live-Server?

Live-server is a tool that will allow you to automatically refresh your browser every time you save any of the files in your projects directory. Yes, it's awesome!

Of course, if you already have a way of live reloading your browser then skip to the code below. Any live reloading will work just fine.

Live-server is a node package that can be installed via NPM. If you do not have node installed head over to nodejs.org and install it on your machine.

Once you have node installed you can run the following command from cmd or terminal:

```
npm install -g live-server
```

That will install live server globally on your machine. Once that is done navigate to your project's directory in terminal and run the following line:

```
live-server
```

Yep, it’s that simple. Live-server will now scan your files for changes and when it sees one it will refresh your browser so you don’t have to.

Your First Bit Of Angular

There is one bit of code that every single Angular application needs, and that is the ng-app directive.

In Angular, a directive is pretty much just a bit of code that tells Angular it needs to do something. Of course it is a bit more complex than that in reality. But that is enough for you to understand right now.

What the ng-app directive does, is tell Angular which parts of the HTML it is in control of. In our case we want it to control our entire page, so we will add it onto our HTML tag.

```
<html lang="en" ng-app="turtleFacts">
```

As you can see, it looks very much like a normal HTML attribute – that’s because it is. Inside the quotes for the ng-app directive, we give our application a name.

The name we give it here is important because that is the name we will be referencing when we start writing the Javascript to control our application.

turtleFacts Angular Module

To create the Javascript to control the page, we first need to reference it in our HTML. The first bit of Javascript we are going to write is to define the module – which is effectively our application.

```
    <!-- Our application scripts -->
    <script src="js/app.js"></script>

</body>
</html>
```

We will add it into a file called app.js inside our js/ folder.

Then we will hop into our app.js folder and start creating the module.

We will start the javascript by creating an Immediately Invoking Function Expression (IIFE)

```
(function(){

    // Angular Code

})();
```

This serves to encapsulate our code. In other words, it keeps our code separate from any other code running on the page and keeps everything clean.

It is ultimately just a function that will run automatically when the page is run. Inside it we will put all of the code we write.

Defining the Module

To define the module we need to grab hold of the Angular object then call .module and pass into it the name of the module and any dependencies.

```
(function(){

    /*
     * Declaration of main angular module for this application.
     *
     * It is named turtleFacts and has no dependencies (hence the
     * empty array as the second argument)
     */
    angular
        .module("turtleFacts", [ ]);

})();
```

I have put the .module call on a separate line from the angular call but you do not have to. I have just done it this way as it is the convention for Angular.

Notice that despite having no dependencies, we still have to pass an empty array as the second argument.

This is something that tripped me up when I first started. I assumed as we have no dependencies, we could just leave it blank. This is not the case, though.

In Angular, if we call .module and only pass in a name without the second argument, we are calling that module and asking for it to be returned to us. However, in this instance, we want to create the module.

This is why we need the second argument. It let’s Angular know we are creating a module and not calling one.

AngularJS Controller – Another Directive

Now that Angular knows we are creating a module and we have told it what part of the HTML that module controls we can now start to create the code that does the controlling on behalf of the module – a controller.

In most Angular applications, you will have many controllers. Each in control of a small part of the HTML. This serves a few purposes. It keeps our code clean and easier to understand and it also makes it easier to test.

Our first controller will be in charge of the list of turtles that we saw in the app demo.

So let's create a div and give it the ng-controller directive then tell it which controller we should use to control that particular div.

```
<div ng-controller="listCtrl">

</div>
```

We have called the controller listCtrl in this case. So now let's create another JS file to hold this controller.

We will put the file in directory called controllers/ that will live in our js folder and we will call the file list.js

So our custom scripts area looks like this so far:

```
<!-- Our application scripts -->
   <script src="js/app.js"></script>
   <script src="js/controllers/list.js"></script>

</body>
</html>
```

Why Put The Controller Into A New File?

When we write Angular, we like to separate our concerns. By that I mean we have very defined bits of code that do one thing and one thing only.

A good practice to get into that will help with this separation of concerns is to put all code that does something different into a new file.

A nice by-product of this is that the files will be short and thus easier to read and ultimately easier to maintain.

Our Controller Code

Start off the list.js file with the same IIFE that we have seen before. Then inside that we need to call our turtleFacts module.

We do that in the same way as we created the module, we just leave off the second argument to .module.

Once we have the module returned to us we can chain on other methods. This time we will chain on the .controller method.

The controller method takes two arguments, the name of the controller and the function that holds the code for the controller.

```
(function(){

    angular
        .module("turtleFacts")
        .controller("listCtrl", ListController);

    function ListController(){
        // List Controller Logic
    }

})();
```

In this case I have decided to use a named function as the second argument to .controller then define the function explicitly below.

Another Way – Although Not My Favourite

Although it is pretty common to use an inline anonymous function as the second argument like this:

```
(function(){

    angular
        .module("turtleFacts")
        .controller("listCtrl", function(){
            // List Controller Logic
        });
})();
```

This method is perfectly valid. Feel free to use it. However, I do not like doing it this way because it can make the code slightly harder to read.

I like to be as explicit and clean as possible. So I find directly naming the function and explicitly declaring it is more natural for me. But you do which ever way you prefer.

The Famous {{}} Angular Syntax

The first thing I ever saw of Angular was a line of code in an HTML file that used the {{}} syntax. This is used to tell Angular we are using some sort of expression and not a literal string as it would usually be in an HTML document.

So something like `{{2+4}}` will be evaluated as an expression by angular and the result will be the number 6 printed out to the screen.

When inside our controller HTML we can take advantage of this {{}} syntax to bind on to properties that we define in our controller.

The great thing about this is that if we then change that property on our controller at some point during the lifetime of the application the value on the screen will automatically update without us having to do anything. This is part of the beauty of Angular data binding.

Chapter 2

Using An Angular Controller To Add Content

In the last part we wrote our first bits of Angular code. One of those bits was the code that instantiates the controller for our list view. In this part we will take that Angular Controller and use it to dynamically insert data into our HTML. This gives us great control over the content that is on our page.

As the saying goes, there are many ways to skin a cat. What that means for us now is that there is more than one way that we can create properties on our controller and insert that data into our HTML.

In this article I will explain the two main methods of doing this and the pros and cons of both. I will of course also tell you which method I prefer and why.

If you want to see the app for yourself, check it out here.

The git repo can be found here.

Angular Controller Video Tutorial

As always, the more visually inclined can just watch this video and you will receive all the same information as you would from the article. If you prefer to read just scroll down past the video.

Click the image to go to the video on youtube or go to https://www.youtube.com/watch?v=mCDI3ZH3E58&index=2&list=PLqr0oBkln1FBmOjK24_B4y_VAA8736wPq

The next part can be found here

Method 1: $scope Service

The first method and the most commonly used method in beginners tutorials is using an Angular Service (more on what these are later in the course) call $scope. Although it is the most common, in my opinion it is not the best method. But I will explain it anyway.

We start off inside the list controller that we created in the previous tutorial and into the function we pass **$scope** . We can view $scope as simply an object that we are passing into our function. We can then attach properties onto that object and have access to those properties in our HTML.

So for example we could attach a property called "dummyData" onto $scope like this:

```
function ListController($scope){
    $scope.dummyData = "Hello  World";
}
```

Heading back into the HTML we could use that wonderful {{}} syntax to grab hold of that property like this:

```
<div ng-controller="listCtrl">
    {{dummyData}}
</div>
```

This will display the text "Hello World" out inside the div. Fantastic.

The Problems With $scope.

This approach is fine in smaller applications but as your applications grow just having bindings to a property like "dummyData" may get confusing. Especially if you have a property called dummyData inside more than one of your controllers, all of which have a different value.

This lack of explicit declaration of the data you are using can become a problem. I prefer to make everything explicit and immediately obvious what I am trying to do.

This is where the next method of doing this comes in.

Method 2: Controller As Syntax

The next method removes the $scope from our function and instead we bind our properties onto the "this" object inside our function.

To make this easier I usually set "this" equal to a variable at the start of the function and use that variable throughout. This saves some potential confusion later on.

```
function ListController(){
    var vm = this;

    vm.dummyData = "Hello World";
}
```

I used the variable name "vm" which simply stands for "view model". We are attaching these properties onto the view model – the view being our HTML.

We then attached that same dummyData property onto vm like we did with $scope earlier.

Our Code Is Broken!

If you now try to render the HTML using this controller code you will find the code no longer works. This is because we have to make a few changes in our HTML. This is where the name of the "Controller As" syntax will become apparent.

We need to make a few small changes. The first of which is changing **ng-controller="listCtrl"** to **ng-controller="listCtrl as list"** .
What we are doing here is creating an alias for our controller. Notice is is the name of the controller as before, then "as list".

So we are referring to our controller as list. In other words, when we use the name “list”, Angular will know we are referring to the listCtrl controller.

Now inside the {{}} we can refer to the exact controller which the dummyData property is on:

```
<div ng-controller="listCtrl as list">
    {{list.dummyData}}
</div>
```

Ahhhh! Yes! Everything is explicit now. No more potential confusion. When we see list.dummyData there is no doubt at all as to where the dummyData property is coming from. Just how we like it.

Which To Use?

Ultimately It does not really matter which of these methods you use. Just pick one and be consistent with it. Consistency is the key when you are coding, especially if you are working in teams.

However, if I was to recommend a method to use, I would definitely recommend going with the controller as syntax. It may be a bit more typing in the short term, but it will save you a lot of headaches in the future.

Chapter 3

Ng-Repeat Directive Will Do Your Work For You

Now armed with the ability to create controller properties with some data attached then inserting that data into our HTML we can move on to create something useful. We will be using the ng-repeat directive (which will save us a lot of typing. YAY!) along with bootstrap to start creating the list of turtles in our application.

Of course, in any application some data is needed. This data is usually fetched from anAPI on a backend server. However, for this tutorial we will just be focusing on the front end implementation of this application. As a result, we will need to simulate the data in some way.

If you want to check out the finished app, you can see it here.

The git repo can be found here.

Angular Video Tutorial Goodness

As always, for those of you more visually inclined this full tutorial is in video form below. For the rest of you continue reading below the video for the same content in written form.

Click the image to go to the video on youtube or go here https://www.youtube.com/watch?v=iAX67gisQ2M&list=PLqr0oBkln1FBmOjK24_B4y_VAA8736wPq&index=3

The next part of the series can be found here.

The way we will simulate the API request in this application is to simply paste the JSON data we would normally get from a server, straight into our code as a variable. We will then attach that variable as a property on our controller which will give us access to all of that data inside our HTML.

This variable will go inside our IIFE for the list controller but it will live outside of the function for the controller itself. We will then create a property on our controller that binds to this.

```
var turtlesData = [
        {
            type: "Green Turtle",
            image_url: "http://www.what-do-turtles-eat.com/wp-content/uploads/2014/10/Sea-Turtles-Habitat.jpg",
            locations: "Tropical and subtropical oceans worldwide",
            size: "Up to 1.5m and up to 300kg",
            lifespan: "Over 80 years",
            diet: "Herbivore",
            description: "The green turtle is a large, weighty sea turtle with a wide, smooth carapace, or shell. It inhabits tropical and subtropical coastal waters around the world and has been observed clambering onto land to sunbathe. It is named not for the color of its shell, which is normally brown or olive depending on its habitat, but for the greenish color of its skin. There are two types of green turtles–scientists are currently debating whether they are subspecies or separate species–including the Atlantic green turtle, normally found off the shores of Europe and North America, and the Eastern Pacific green turtle, which has been found in coastal waters from Alaska to Chile."
        },
        {
            type: "Loggerhead Turtle",
            image_url: "http://i.telegraph.co.uk/multimedia/archive/02651/loggerheadTurtle_2651448b.jpg",
            locations: "Tropical and subtropical oceans worldwide",
            size: "90cm, 115kg",
            lifespan: "More than 50 years",
            diet: "Carnivore",
            description: "Loggerhead turtles are the most abundant of all the marine turtle species in U.S. waters. But persistent population declines due to pollution, shrimp trawling, and development in their nesting areas, among other factors, have kept this wide-ranging seagoer on the threatened species list since 1978. Their enormous range encompasses all but the most frigid waters of the world's oceans. They seem to prefer coastal habitats, but often frequent inland water bodies and will travel hundreds of miles out to sea."
        },
        {
            type: "Leatherback Turtle",
```

So here we have simply created a turtlesData variable and set it equal to all of the JSON. The JSON itself is all the information for all of our turtles. There is a name, image, location, size, lifespan, diet and general description for each turtle.

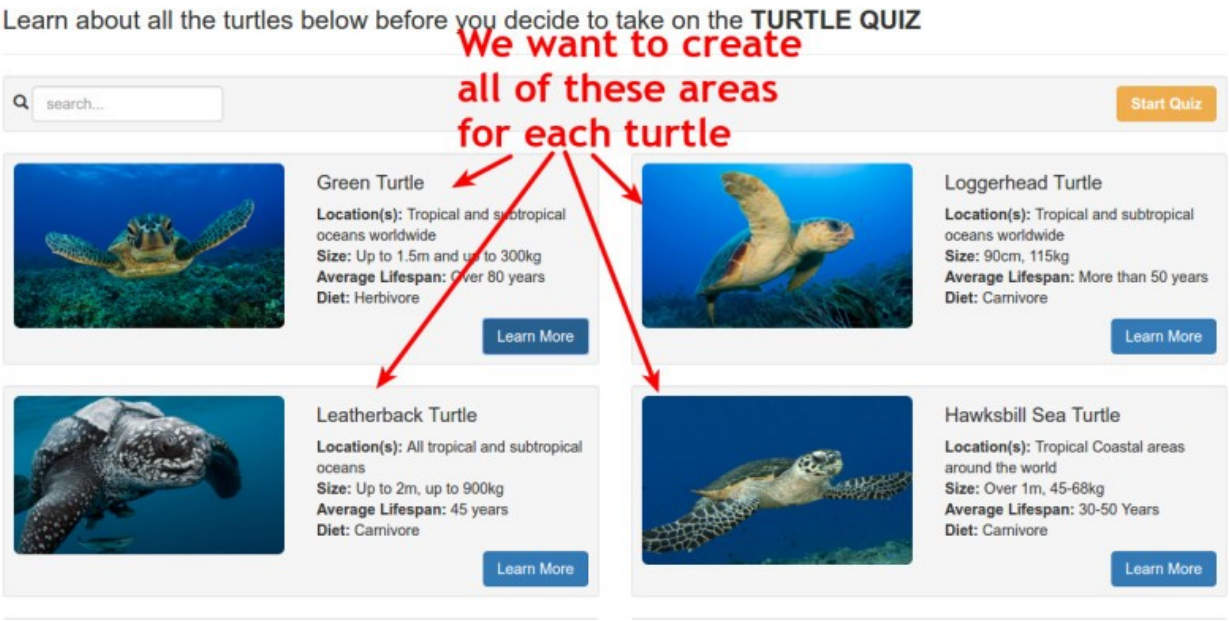
Now we can create the property inside our controller function. That is as easy as creating a property and setting it equal to the variable we created called turtlesData.

```
function ListController(){
    var vm = this;

    vm.data = turtlesData;
}
```

This is because our controller function and the variable are within the same context in the code ie. they are both inside the same IIFE. (Don’t worry if you don’t understand that last sentence. It isn’t super important right now. Although, I will do a tutorial explaining all of that sort of stuff soon).

What is this Ng-Repeat Directive You Speak Of?

Now that we have access to the information about all of our turtles we can start thinking about how to create all of our markup.



The old fashioned way of doing that would be to hard code all of the HTML for each of our turtles individually. Of course this results in a lot of repeated code and waaaaaay too much typing for us to do. We are lazy remember!

In steps the ng-repeat directive. What this lovely directive allows us to do is declare an area of markup and tell Angular that it should repeat that markup for each item in a dataset that we specify.

In our case we will create the markup for one of our turtles – also taking advantage of the {{}} binding to grab hold of each bit of information we want. Then using the ng-repeat directive, we will tell angular to simply repeat all of that markup for each turtle in the JSON. Phew! That's more like it.

We Need An Alias Again.

Much like we used an alias for our controller when we used the controller as syntax, we will also use an alias when using ng-repeat. The alias in this case will be the name we will use to reference each iteration of the repeating loop through our data.

The markup for a general ng-repeat will look something like this:

```
<div ng-repeat="data in controller.items">
    {{data}}
</div>
```

Here we have a property on our controller called items which is an object or array that we can loop through. We give each iteration through that loop an alias of data.

Using that alias we can put inside the {{}} syntax and that will print out the value of each respective value in the items array or object.

Of course, we can also harness this if our data is an object or array or objects or array ie nested objects or arrays (or multidimensional would be another name). This is in fact what we are going to do.

The Markup For Our List of Turtles.

The top level property we are going to loop through is the data property on our list controller which of course is our JSON data. Each iteration through the ng-repeat will loop through each turtle. But each turtle has many properties such as type, image_url, location etc.

We can grab hold of these by using the dot notation something like this:

```
<div ng-repeat="turtle in list.data">
    {{turtle.type}}
</div>
```

This will print out the type property of each turtle in our data property. Using this we can now build the actual markup we need and lay out our information for each turtle nicely.

Bootstrapping Our App Markup With Bootstrap

This course is not focused on Bootstrap so I will not spend a lot of time explaining the different elements used. However, if you want to learn more let me know and I may do a full course on Bootstrap. Especially with Bootstrap 4 looming just around the corner.

Here is the start of the markup with the ng-repeat directive added in:

```
<div class="row">
    <div class="col-sm-6" ng-repeat="turtle in list.data">

    </div>
</div>
```

The class we have given the inside div here is what will make the element responsive to different screen sizes. The ng-repeat has been added to the inside div so that div and anything inside it will be repeated for every element in the data property on the list controller.

We now want to create the grey area that will contain the turtles themselves then add the image and all the info for each turtle respectively.

This is the Bootstrap to do that along with the bindings to the data that we need:

```
<div class="row">
    <div class="col-sm-6" ng-repeat="turtle in list.data">
        <div class="well well-sm">
            <div class="row">
                <div class="col-md-6">

                    <img ng-src="{{turtle.image_url}}"
                        class="img-rounded img-responsive well-image">
                </div>
                <div class="col-md-6">
                    <h4>{{turtle.type}}</h4>

                    <p><strong>Locations: </strong>{{turtle.locations}}</p>
                    <p><strong>Size: </strong>{{turtle.size}}</p>
                    <p><strong>Average Lifespan: </strong>{{turtle.lifespan}}</p>
                    <p><strong>Diet: </strong>{{turtle.diet}}</p>

                </div>
            </div><!-- row -->
        </div><!-- well -->
    </div><!-- col-xs-6 -->
</div>
```

ng-src? Wait! That's new!

This should all be easy to understand for the most part. The one new thing that you haven't seen yet is the ng-src on the image. Notice that the image does not have a normal src attribute at all.

The reason for this is because the URL for the image is coming from the JSON data and therefore we want to use the {{}} binding syntax to grab hold of it.

So you may think that we could just add that binding syntax to the normal src attribute and Angular will replace the binding with the url and the image will render just fine.

Unfortunately this is not the case. The reason for this is easy to understand though.

Basically, it comes down to the order things are rendered. With images when you use the normal src attribute the HTML tries to fetch the URL straight away as the page is loading. Importantly, this is before Angular has had time to load and hook into the page.

This of course means that the HTML will see {{turtle.image_url}} when it looks at the src attribute. This literal string of course isn't a URL and the HTML will not know what to do with it.

Finally when Angular loads it will change that literal string to the URL that we want but by that stage the HTML thinks all of the work with image urls is done and it will not try to fetch the url and therefore no image will be displayed.

To stop this problem, Angular added in a helpful directive called ng-src which is to be used in place of the normal src attribute when dealing with Angular model data.

Now the HTML doesn't see a normal src attribute so it will simply skip that image. But when Angular loads up and hooks into the page it will see the ng-src and fetch the image from the url that it obtains from our data and the image will now display as we want.

Finishing Off The List Markup

The final thing we need to do is add the markup for the “Learn More” button. We will be using simple Bootstrap classes to style the button. The markup itself will go directly after the paragraphs that contain all the turtle information.

We will need to used two Bootstrap attributes on this button as will intend this button to trigger the modal that we will create later.

```
<button class="btn btn-primary pull-right"
    data-toggle="modal"
    data-target="#turtle-info">Learn More</button>
```

The data-toggle bootstrap attribute let’s bootstrap know that we intend to use this button to trigger a modal.

The data-target bootstrap attribute tells bootstrap exactly which modal we want to trigger with this button. In this case we have told it will be the modal with the id of “turtle-info”. We will need to remember that and give the modal that id when we create it.

Chapter 4

Bootstrap Modal Markup And Other CSS

We made a good start on our bootstrap markup in last part but we did end up leaving a few little issues with the CSS most notably the image sizing was not consistent. So let's kick things off this time by just fixing up those problems. Then we will start creating the bootstrap modal.

If you want to see the app in action, check it out here.

The git repo can be found here.

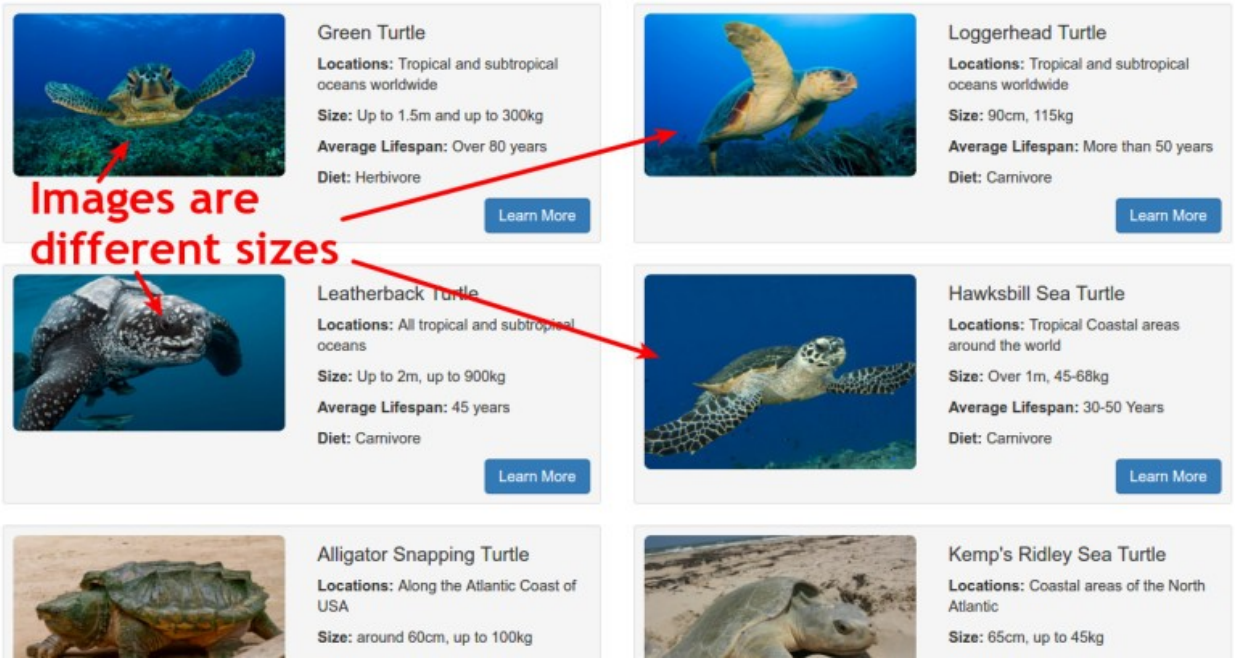
As Always – Video Or Text

You can watch this full tutorial in video form below or you can continue reading past the video for a full written version.

Click the image to go to the video on youtube or go here https://www.youtube.com/watch?v=IW37XF2g7AU&index=4&list=PLqr0oBkln1FBmOjK24_B4y_VAA8736wPq

The next part can be found here.

Fixing Image Sizing Issues



We will get ourselves ready for resizing images by adding a class to each of the image tags in our HTML.

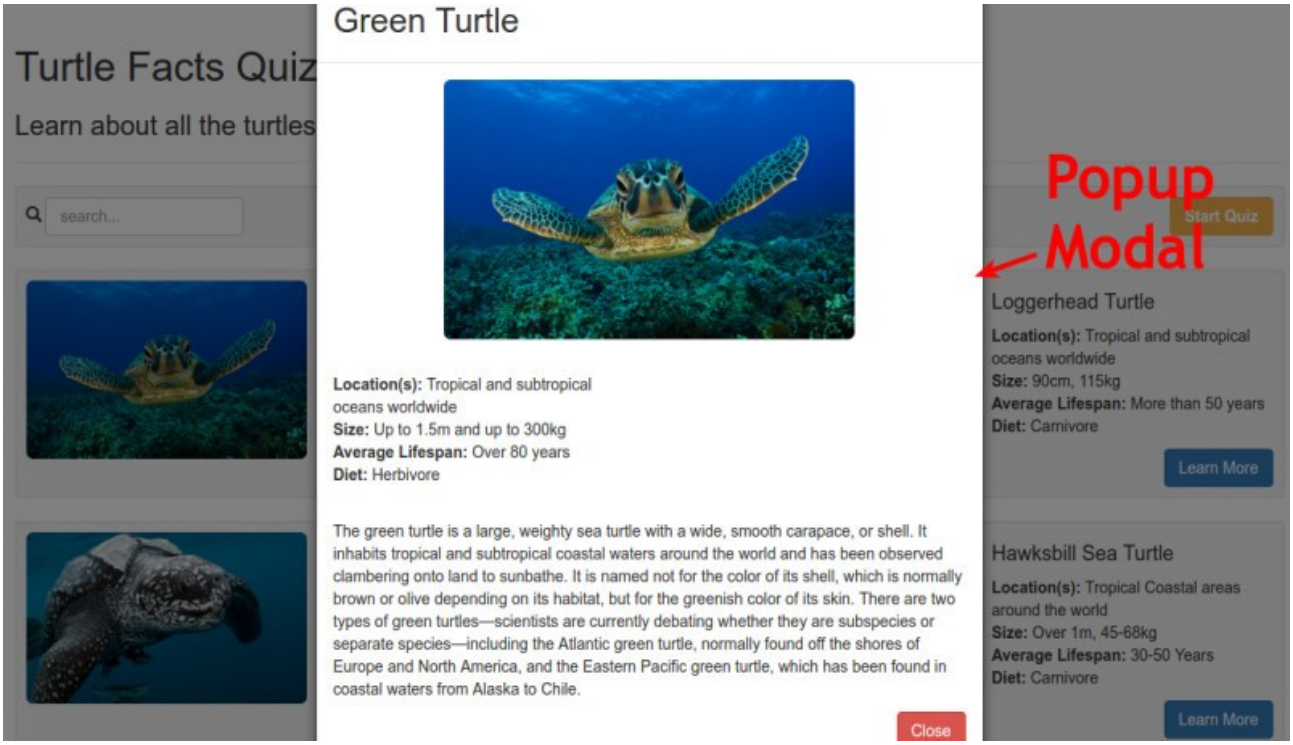
```
<img ng-src="{{turtle.image_url}}"
     class="img-rounded img-responsive well-image">
```

The class we use here is well-image, but we could use anything that we want here. Now, inside our style.css file which lives in our CSS folder we can create some rules for our well-image class.

```
.well-image{
    width: 100%;
    height: 162px;
}
```

We've given to the width of 100% just to ensure that the images always the full width of its container. The height could be any value that you choose I've just decided on 162 pixels due to some trial and error.

Let's Make The Modal Pop Up!



We want the modal to pop up on the screen when the user clicks the "Learn More" button for any one of the turtles. This will be achieved using some functionality that is built into bootstrap. We will also need to trigger some controller functionality when the button is clicked and we will do that with a new Angular directive called ng-click.

We will add this new ng-click directive to the button along with the data-toggle and data-target attributes that we added in the previous part. Inside the quotes for the ng-click we give it the function that we want to trigger on the click event.

```
<button class="btn btn-primary pull-right"
        data-toggle="modal"
        data-target="#turtle-info"
        ng-click="list.changeActiveTurtle(turtle)">Learn More</button>
```

In this case we have called the function changeActiveTurtle, into which we pass turtle, which if you remember from the previous part is the alias given to each iteration of the ng-repeat.

So what this means for us is that when we click on the learn more button on any one of the individual Turtles the function will be passed the data for that particular turtle.

The function will then use this turtle information passed to it by the ng-click event to manipulate a property on our controller which indicates the currently active turtle.

It will then be the data associated to this currently active turtle property that we use to populate the modal with data. We will of course, have to create this function later.

Inside the Angular Controller – Moar Properties

Before we create the function that manipulates the active turtle property we have to actually create that property on our controller. As there is nothing active when the application first loads we will just initialise this property to an empty object.

We use an object because the property will eventually be set to one of the sets of turtle data in our JSON, which are obviously objects.

```
function ListController(){
    var vm = this;

    vm.data = turtlesData;
    vm.activeTurtle = {}; // will be used in the view to hold the data of currently activ

}
```

Now I need to create a function that will manipulate this active turtle property. Again, it would be perfectly fine to declare this function and set it equal to an anonymous function something like this.

```
function ListController(){
    var vm = this;

    vm.data = turtlesData;
    vm.activeTurtle = {}; // will be used in the view to hold the data of currently acti

    //This is one way you could declare the function, but not my preferred method
    vm.changeActiveTurtle = function(){
        // Code Here
    }

}
```

However, as you already know I prefer to declare the function by setting it a property equal to a named function and then declaring that named function further down in our controller. Something like this.

```
function ListController(){
    var vm = this;

    vm.data = turtlesData;
    vm.activeTurtle = {}; // will be used in the view to hold the data of currently acti

    // This is my preferred method
    vm.changeActiveTurtle = changeActiveTurtle;

    function changeActiveTurtle(){
        // Logic in here
    }
}
```

So now let’s create the logic inside the changeActiveTurtle function. When we called the function in the ng-click we pass today in the value of turtle which is the current index of the button that we clicked on and the data associated with that index. So we will need to give our functions in the controller some arguments.

The actual logic inside the function is only a single line of code that grabs hold of the activeTurtle property from within our controller and set equal to the index data that we passed into the function.

```
function ListController(){
    var vm = this;

    vm.data = turtlesData;
    vm.activeTurtle = {}; // will be used in the view to hold the data of currently acti

    // This is my preferred method
    vm.changeActiveTurtle = changeActiveTurtle;

    function changeActiveTurtle(index){
        vm.activeTurtle = index;
    }
}
```

HTML Markup For The Bootstrap Modal

Now every time the learn more button is clicked the ng-click directive will trigger the changeActiveTurtle function which will set the active turtle to an object that contains all of the data associated with the turtle the user wants to learn more about.

We can now use this data in the modal. This allows us to create a generic modal with placeholders using the {{}} Angular syntax that will then just insert the active turtle data as and when a particular turtle is clicked on.

So now it's time to create HTML for this bootstrap modal and insert the angular bindings into it. We'll start off by just creating a div for the modal and giving it the ID that we referenced in the data-target attribute earlier – turtle-info. Along with this will also add a few more divs which are required by bootstrap to create a modal.

```
<div class="modal" id="turtle-info">
    <div class="modal-dialog">
        <div class="modal-content">
            <div class="modal-header">

            </div>
        </div>
    </div>
</div>
```

A Little Reminder About The JSON

At the top of the modal area we want to display the name of the turtle that is being displayed. We will add this into an H2 tag inside a modal header div. Inside the H2 is where we will use the {{}} syntax to grab hold of the name of the currently active turtle.

Because the active turtle object is just one of the objects inside the JSON which looks something like the below snippet. Using the dot operator on the activeTurtle object, we are able to grab hold of the type, the locations, the image_url etc.

```
{
    type: "text",
    text: "Where does the Kemp's Ridley Sea Turtle live?'",
    possibilities: [
        {
            answer: "Tropical waters all around the world"
        },
        {
            answer: "Eastern Australia"
        },
        {
            answer: "Coastal North Atlantic"
        },
        {
            answer: "South pacific islands"
        }
    ],
    selected: null,
    correct: null
}
```

So now with the H2 added into our modal the code looks like this.

```
<div class="modal-header">
    <h2>{{list.activeTurtle.type}}</h2>
</div>
```

When I want some responsive code that will centre a large image of the turtle inside the mobile. We will make a div that is 8 columns (out of the possible 12) wide and then offset it by two columns to give it to columns of space on each side to make up the full 12 and that the div is centred.

```
<div class="modal-content">
    <div class="modal-header">
        <h2>{{list.activeTurtle.type}}</h2>
    </div>

    <div class="model-body">
        <div class="row">
            <div class="col-xs-8 col-xs-offset-2">
                <img ng-src="{{list.activeTurtle.image_url}}"
                     class="img-rounded img-responsive">
            </div>
        </div>
    </div>
</div>
```

The code inside the ng-src is exactly the same as what we've seen earlier except we are referencing the image_url on the activeTurtle property instead.

Adding The Text Info About The Turtle

To finish off the modal markup we want to create the area that will hold all of the text data about our turtle. This code should look very familiar to you so I won't explain it in depth.

```
<div class="modal-content">
    <div class="modal-header">
        <h2>{{list.activeTurtle.type}}</h2>
    </div>
    <div class="modal-body">
        <div class="row">
            <div class="col-xs-8 col-xs-offset-2">
                <img ng-src="{{list.activeTurtle.image_url}}" class="img-rounded img-re
sponsive">
            </div>
        </div>
        <div class="row">
            <div class="col-md-6">
                <p><strong>Locations: </strong>{{list.activeTurtle.locations}}</p>
                <p><strong>Size: </strong>{{list.activeTurtle.size}}</p>
                <p><strong>Average Lifespan: </strong>{{list.activeTurtle.lifespan}}</p
>
                <p><strong>Diet: </strong>{{list.activeTurtle.diet}}</p>
            </div>
        <div class="col-xs-12">
            <p>{{list.activeTurtle.description}}</p>
        </div>
    </div>
</div>
```

At this point the text to slightly too close to the image and the description is too close to the bullet point so we want to just add a bit of margin at the top of these areas just to make it look a bit nicer.

We will add a class of top-buffer to the row contains all of the text currently. Then we will create this CSS rule.

```
.top-buffer{
    margin-top: 30px;
}
```

The final things to do in our model is to add the close button that will return us back to the list view. To do this will just create a simple button element and add some bootstrap classes to style it and float it right. A bootstrap attribute called data-dismiss is what will actually allow the button to exit the modal.

Chapter 5

Angular Filters Create A Magic Search!

Ok, enough playing around, let's really dig into the power of AngularJS. Creating search functionality from scratch can be notoriously hard, but in this tutorial we will see how easy it is to create an automatically updating search feature using Angular filters.

Using Angular filters in this way is definitely one of my favourite features of AngularJS. It makes seemingly difficult tasks so easy and straight forward to do.

If you want to take a look at this search functionality in action, it can be seen in the video below or check out the app here. As always this video is exactly the same as this article just to give you some options of how you want your information.

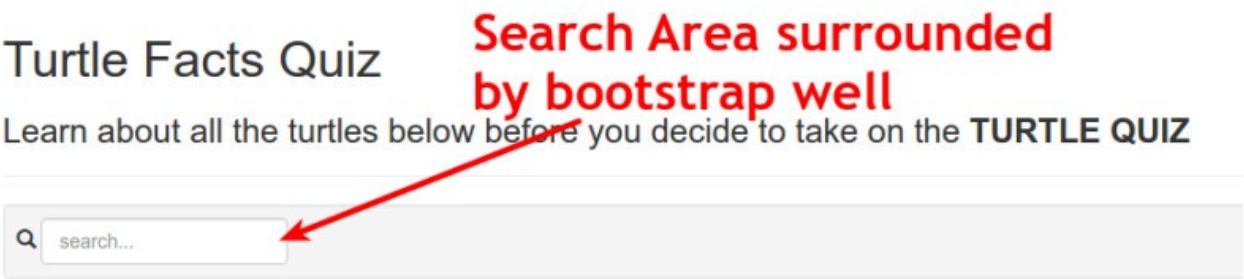
The git repo can be found here.

Video Killed The Radio Stars

Click on the image to go to the video on youtube or go here https://www.youtube.com/watch?v=zisGjJySdLA&index=5&list=PLqr0oBkln1FBmOjK24_B4y_VAA8736wPq

[The next part can be found here]

Getting straight into it, will start off by creating an HTML form which will style with some bootstrap classes. We will add this code right at the top of the HTML markup for our list controller, as we want the search form to be at the top of the page.



This area is the whole grey box at the top of the page that will contain the search box as well as the start quiz button that we will create later. Right now we will focus on creating the search box by adding the icon and the text input area. The icon will be a glyphicon which comes bundled with Bootstrap these days.

```
<form class="form-inline well well-sm clearfix">
    <span class="glyphicon glyphicon-search"></span>
    <input
        type="text"
        placeholder="Search..."
        class="form-control">
</form>
```

The form control class we used on the input is another bootstrap class just to let bootstrap know what this input is doing.

Ng-Model Directive

Now that we have the markup done we can start creating the magic of the search functionality. To start this, we're going to introduce a new directive – ng-model.

What ng-model does it allows us to bind an input on our page directly to a property on the view model. Remember we used the alias "vm" in our controller for the properties that the view model has access to.

So now if we create a property inside the controller and attach it on to the view model we can use ng-model to bind that directly to our input. This means that if we programmatically

change the property inside the controller it will automatically update the input and vice versa.

```
function ListController(){
    var vm = this;

    vm.data = turtlesData;
    vm.activeTurtle = {};

    // Adding the Search property to be used in the ng-model
    vm.search = "";



    vm.changeActiveTurtle = changeActiveTurtle;

    function changeActiveTurtle(index){
        vm.activeTurtle = index;
    }
}
```

Here we created a property called “search” which would just be a string that contains the text the user is searching for. Which of course, is set to an empty string initially. Now let’s add the ng-model directive on to our input and bind to the search property.

```
<form class="form-inline well well-sm clearfix">
    <span class="glyphicon glyphicon-search"></span>
    <input
        type="text"
        placeholder="Search..."
        class="form-control"
        ng-model="list.search">
</form>
```

Using The Search Property With Angular Filters

If you're used to using Unix command line commands then you might be used to using the pipe symbol | to quite literally "pipe" the output of one command into the input of another. This is how Angular filters work.

We want to filter the output of the ng-repeat which is what constructs the entire list of turtles. So inside the quotes of our ng-repeat where it says "turtle in list.data" we had the pipe symbol at the end and then add the filters we want.

There are many filters that are included with Angular and you can also make your own if you want to. But the filter that we're concerned with for the search functionality is called "filter" – the filter filter.

We will also be taking a look at other filters like the number filter later on this course.

The way filters work in angular is you specify the name of the filter you want to use, then a colon, followed by any argument she want to give the filter. In this case the argument we want to give the filter filter is the text that the user is searching for.

```
<div class="col-sm-6" ng-repeat="turtle in list.data | filter:list.search">
```

How Is This Working?

This is a good illustration of the dynamic nature of AngularJS. I mentioned earlier the ng-repeat directive works something similar to a for loop in normal programming languages. But as you can see hear this is not quite right.

In a programming language, when a for loop is finished running that’s it. The ng-repeat and all other directives are constantly updating when you inputs are given.

This is us to use the filter in real time. When we type something into the input comma updates search property in our controller which is what is filtering the ng-repeat. Because this is now changed the ng-repeat will run again and repopulate the list with only entries that satisfy the filter. In other words only with Turtle Data that contain the search term.

Chapter 6

Ng-Click Directive And Ng-Show / Ng-Hide

We're nearly in a position to move away from our list controller and start building out the quiz controller. But before we can do that we need to create the "start quiz" button, which is what we will tackle first. We will then use the ng-click directive to control functionality when the button is clicked.

If you want to see the app for yourself, check it out here.

The git repo can be found here.

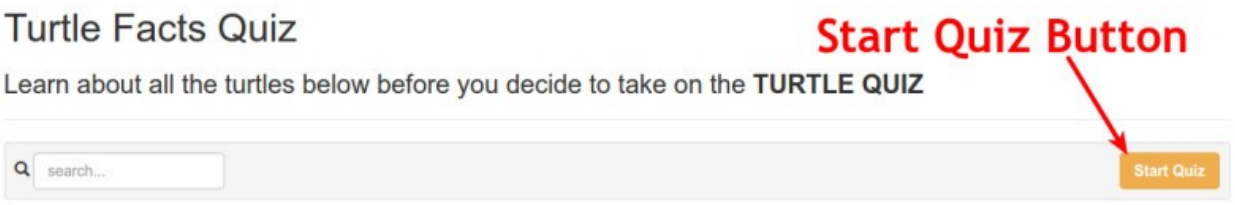
Video Or Text Tutorial?

Take your pick. You can watch the video below or you can read through this article to get the same information.

Click the image to go to the video on youtube or go to https://www.youtube.com/watch?v=ZKbPtYYbSOg&index=6&list=PLqr0oBkln1FBmOjK24_B4y_VAA8736wPq

The next part can be found here

As you can see from the example application in the video the button is on the right hand side of the grey area where are search bar is. This is part of the same form in the HTML. So we will start by creating a button with some bootstrap classes to pull it to the right.




```
<form class="form-inline well well-sm clearfix">
    <span class="glyphicon glyphicon-search"></span>
    <input
        type="text"
        placeholder="Search..."
        class="form-control"
        ng-model="list.search">

    <button class="btn btn-warning pull-right">
        <strong>Start Quiz</strong>
    </button>
</form>
```

Showing And Hiding HTML elements with Angular

Now that we actually have the button we needed to do something. When we click the button we want our turtle list markup to disappear and be replaced with the quiz. To do this we will utilise a partnership between the ng-click directive and ng-hide.

Both of these are new directives but they’re also pretty self explanatory. We will first focus on the ng-show and ng-hide directives.

Ng-hide will accept a boolean or an expression that evaluates to a boolean. If this boolean is true then that element will hide and if it's false the element will show. Ng-show is another similar directive that is the exact opposite of ng-hide. Either can be used but they tend to both be useful to keep things more semantically explicit.

The boolean that we will use to hide the list controller markup, as well as showing the quiz controller markup (using ng-hide and ng-show respectively) will be a property on our controllers which we will call quizActive.

We will add the following code to our list controller.

```
function ListController(){
    var vm = this;

    vm.data = turtlesData;
    vm.activeTurtle = {};
    vm.search = "";

    //Set to false initially so the list shows when the app loads
    vm.quizActive = false;

    vm.changeActiveTurtle = changeActiveTurtle;

    function changeActiveTurtle(index){
        vm.activeTurtle = index;
    }
}
```

Property is initially set to false as when the application loads the quiz is not active and we want the list controller to display. For this reason we can use ng-hide on our list controller div and it will display when the application first loads just like we want (because quizActive is false, so it does not hide the div).

We can add the ng-hide directive to our list controller markup like so.

```
<div ng-controller="listCtrl as list" ng-hide="list.quizActive">
```

But Nothing Happens When I Click The Button!

We can now show and hide the list controller markup (or any html elements for that matter) using Angular. But now we want to programmatically hide the list controller when we click the start quiz button.

This is achieved using the ng-click directive to trigger a function on our controller that will change the quizActive property. Inside the quotes of the ng-click directive we will add the name of a function or literal expression (in our case a function name) that will run when that HTML element is clicked.

```
<button class="btn btn-warning pull-right"
        ng-click="list.activateQuiz()">
            <strong>Start Quiz</strong>
</button>
```

Now we need to create the activateQuiz function. We will do this in the same way that we have created a function in the past – by initialising and named function inside or controller.

The activateQuiz function will be extremely simple as all we need to do is set the quizActive property to true.

```
vm.activateQuiz = activateQuiz;

function activateQuiz(){
    vm.quizActive = true;
}
```

Chapter 7

Angular Services – Serving (Some) Of Your Needs

Fully armed with the knowledge on how to hide and show HTML elements, we can now go ahead and create the quiz controller that we will show at the same time that we hide the list controller – when the user clicks the “start quiz” button. Then we can learn about Angular Services and how they solve some problems that may arise.

If you want to see the app for yourself, check it out here.

The git repo can be found here.

Pick Your Poison – Text Or Video

As always, you can read this article or you can watch the video below to get the same information.

Click the image to go to the video on youtube or go to https://www.youtube.com/watch?v=HrbkZO5Mt0g&index=7&list=PLqr0oBkln1FBmOjK24_B4y_VAA8736wPq

The next part can be found here

Below our list controller markup in the HTML we can create new controller div for our quiz. Just like we did with our list controller we will use the “controller as” syntax.

```
<div ng-controller="quizCtrl as quiz">
    <!-- Rest of the markup goes here -->
</div>
```

Of course, we will add this controller into another JavaScript file in our controller directory. So we will need to reference that script in the HTML. So are application scripts at the bottom of our HTML looks like this now.

```
<script src="js/app.js"></script>
<script src="js/controllers/list.js"></script>
<script src="js/controllers/quiz.js"></script>
```

Let's Create Another Angular Controller

To start off the controller, we will use the immediately invoked function expression (IIFE)that we have seen in earlier parts. The rest of the setup will be familiar as well. The only change is the name of the controller that we create.

```
(function(){

    angular
        .module("turtleFacts")
        .controller("quizCtrl", QuizController);

    function QuizController(){

        var vm = this;

    }

})();
```

How Do We Show the Quiz Controller With the Quiz Is Active?

At this point, you may or may not have realised that we are running into a problem. Hiding the list controller was simple, the quizActive flag is on the list controller, so we can manipulate it and then use it for the ng-hide.

However, we want to show the quiz controller at the same time. So ideally, we would use the same quizActive property to do that right? quizActive flicks to true and the ng-hide on the list controller hide the list controller and we use ng-show on the quizController to show that at the same time.

Unfortunately, Angular does not allow controllers to communicate like that. Controllers are totally separate. So we do not have access to the quizActive flag on the list controller when we are inside the quiz controller. So what shall we do?

Introducing Angular Services

This is where we can use a useful tool in the Angular Toolbox – services. There are many types of services available in Angular. Too many to cover in this tutorial series. So for now, we will be focusing on one type of service called Factories.

Factories in Angular will allow us to share some data between the controllers – which is exactly what we want.

A common use case for services like factories in angular would be for example, an application with multiple controllers, all of which need to pull some data from an API. In that case, it would be possible to write the API calls into every controller separately. But this is repeating code and breaking a key rule of code.

Instead, that API logic could be put into a service and all controllers can be given access to that service. Note that a controller wouldn't automatically have access, you would have to explicitly allow it.

Now, all the controllers can make an API call without having to repeat code. Perfect. This also serves to separate out concerns and ensure that all bits of code are only doing one thing. Added bonus.

Because controllers can share code via a service, it is possible to use the service to contain properties that all the controllers will need access to. This is our use case.

Get Building Your Factory!

We will create a new directory in our project called factories, which is, of course, where we will store any factories we create. We can then create a new script called quizmetrics.js which will be a factory that holds all data relevant to the general operation of the quiz app that all controllers may need access to.

Again, we use an IIFE to start things off. Then call our turtleFacts module. But this time we call a new method – .factory. The factory method is much like the controller method in that it accepts two arguments – the name of the factory and the function that defines the factory.'

```
(function(){

    angular
        .module("turtleFacts")
        .factory("quizMetrics", QuizMetrics);

    function QuizMetrics(){


    }

})();
```

A Brief Aside About How Factories Are Built

A factory does some logic then it returns something. It could be an object, or just a single property or a function or anything. Whatever it returns, is what the controllers that use the function will have access to.

So for our purposes, we will build an object that contains many properties that we will need in our controllers. Then this object is what we will return from the factory and ultimately will be what is shared between the controllers.

We will start creating the object inside the factory by adding the quizActive property that is currently in our list controller. We can then remove it from the list controller once that is done.

```
function QuizMetrics(){

    var quizObj = {
        quizActive = false
    };

    return quizObj;

}
```

Don’t forget to add the script that references the factory to the end of the HTML. I placed the code in js/factories/quizMetrics.js, so that is the source in my HTML script tag.

Chapter 8

Make The Factory Useful – Dependency Injection

It is no use to us to have a factory if we cannot use it inside our controller. In this part we will inject the factory into the list controller so we can use it. Then we will refactor the list controller to remove the now redundant quizActive code. Let's dive into some dependency injection.

If you want to see the app for yourself, check it out here.

The git repo can be found here.

You Know The Drill.

Watch the video, or read the article. Or both.

Click the image to go to the video on youtube or go here https://www.youtube.com/watch?v=txzLabEIP_w&index=8&list=PLqr0oBkln1FBmOjK24_B4y_VAA8736wPq

[The next part can be found here.]

First things first, injecting the factory into the controller. To do this we will make use of an Angular method called $inject. This is an explicit way for us to inject any dependencies into our controllers.

```
(function(){

    angular
        .module("turtleFacts")
        .controller("listCtrl", ListController);

    ListController.$inject = ['quizMetrics'];

    // Rest of code left out to keep this snippet clean

})();
```

As you can see, with $inject we call it and set it equal to an array. Inside this array we just list all the dependencies in standard array syntax.

```
(function(){

    angular
        .module("turtleFacts")
        .controller("listCtrl", ListController);

    ListController.$inject = ['quizMetrics'];

    function ListController(quizMetrics){
        var vm = this;

        vm.quizMetrics = quizMetrics;

        // Rest of Controller code
    }

})();
```

We are not done quite yet. We now need to pass that factory that we injected, into our controller function as an argument. This allows us to then bind to the properties on the factory like we would with any other properties.

We have given the view model (ie. the html) access to all properties on the factory object by creating a property on the controller using the vm. Syntax and setting it equal to the factory object.

Getting Rid Of The Redundant Code

We now have access to the quizActive property on the factory but also have a quizActive property inside the controller. This is now redundant so we shall remove it.

There is also a function in the controller called activateQuiz. But isn’t this logic part of the general logic for the quiz? It is not specifically for the list controller.

This means that the list controller is doing more than one job! No separation of concerns. So we will remove that function from the controller and add that function into our factory, thus allowing all controllers access to that code.

With that code removed from the list controller, the controller looks like this:

(Notice we haven't removed the activateQuiz function, just the logic inside it. This is because we will call the function we add to our factory from inside this function in our controller later on)

```
function ListController(){
    var vm = this;

    vm.data = turtlesData;
    vm.activeTurtle = {};
    vm.search = "";
    vm.changeActiveTurtle = changeActiveTurtle;
    vm.activateQuiz = activateQuiz;

    function changeActiveTurtle(index){
        vm.activeTurtle = index;
    }

    function activateQuiz(){
        // Fix this further down the tutorial
    }
}
```

Now let's add that function that we removed from the controller into the factory. To keep things more general (and maybe help us down the road) we will call the function in the factory changeState.

```
(function(){

    angular
        .module("turtleFacts")
        .factory("quizMetrics", QuizMetrics);

    function QuizMetrics(){

        var quizObj = {
            quizActive: false,
            changeState: changeState
        };

        return quizObj;

        function changeState(state){
            quizObj.quizActive = state;
        }
    }
}
```

Wait! That Looks Weird!

Now, you may notice a few things with that above snippet from our factory. Number one, we are again using named functions instead of inline anonymous functions. Again, this keeps things more explicit.

But the second thing you may notice, especially if you are used to more traditional programming languages like C, is that we are declaring our functions at the bottom, below the return statement. This would be invalid code in many languages. So why do we do it here?

First of all, JavaScript allows this syntax due to a concept called hoisting, which I won't go into now. Just know that the code is valid. Secondly, because it is valid code, we do it this way to make things easier to read. To separate out the interface and the implementation.

So when you first see this factory, you will straight away see the object that is being returned. At a quick glance you can know exactly what the code is doing at a high level, without having to dig into the actual implementation. It just makes it easier for us developers to read. The computer doesn't care.

Back In The Controller

Inside the activateQuiz function that we left blank earlier we can now call the changeState function that is on our factory and pass it in the value of true. This will set the quizActive to the true state.

```
function activateQuiz(){

    // We will create this function on the factory soon
    quizMetrics.changeState(true);
}
```

You may be wondering why we called quizMetrics.changeState and not vm.quizMetrics.changeState.

The reason for this is that we passed in quizMetrics to the controller, so we still have access to it in it raw form. We just added the used the vm. Syntax to attach all the quizMetrics properties to our view. But it is a secondary source. We should still be using the original factory that was passed in.

Back To The Future – Now With Dependency Injection

We are nearly back to the functionality we had before we started playing around with the factory. The only thing that is left to change is the ng-hide on the list controller html. This is still referencing list.quizActive, which doesn't exist anymore.

```
<div ng-controller="listCtrl as list" ng-hide="list.quizMetrics.quizActive">
```

Dependency Injection For The Quiz Controller

Now we inject the factory into our quiz controller in the exact same way as we did with the list controller. The quiz controller script will look something like this:

```
(function(){

    angular
        .module("turtleFacts")
        .controller("quizCtrl", QuizController);

    QuizController.$inject = ['quizMetrics'];

    function QuizController(quizMetrics){

        var vm = this;

        vm.quizMetrics = quizMetrics;

    }

})();
```

Now we have access to the factory properties – including the quizActive property. We can now use this in an ng-show to show the quiz controller markup whenever the start quiz button is pressed.

```
<div ng-controller="quizCtrl as quiz" ng-show="quiz.quizMetrics.quizActive">
```

Chapter 9

Mocking An API Request With Angular Factories

We have already covered how to build a basic factory when we built the quizMetrics factory in a previous part. In this part, we will build another factory; this time to mock data coming from an API. Let's continue using Angular factories to further separate concerns.

Earlier, we mentioned that we won't be using an actual API in this application but will instead copy and paste the JSON into the scripts. This is what we did for the list data in the list controller.

The only problem with that is that now we have the JSON copied into the list controller and now if we followed that same idea we would copy the quiz JSON into the quiz controller. Things are starting to get messy.

To solve this problem we will create a factory and copy all of the data into that instead (including refactoring the data out of the list controller and into the factory). That way our controllers have no knowledge of where the data is coming from, just that it is getting the data.

This way, we can copy the data in for now, but at a later date we can actually make API requests and put all the logic into the factory and the controller still receives the data in the exact same way. We won’t have to touch the controllers to make this change. Can you see how this allows our application to grow over time?

We don’t have to completely refactor the entire codebase to allow API calls, which is something we would have to do if we copied the data into each controller separately. We now put all that data into a factory and separate all of our concerns. It’s all coming together nicely.

If you want to see the app for yourself, check it out here.

The git repo can be found here.

Get On With It!

Ok, enough chat, let’s write some code.

Click the image to go to the video on youtube or go here https://www.youtube.com/watch?v=TlR3bI7Azvk&list=PLqr0oBkln1FBmOjK24_B4y_VAA8736wPq&index=9

[The next part can be found here]

We will create another script in the factories directory and call it dataservice.js. We will start this in the exact same way we started the quizMetrics factory.

```
(function(){

    angular
        .module("turtleFacts")
        .factory("DataService", DataService);


    function DataService(){

        var dataObj = {
            // We will add properties to this object soon
        };

        return dataObj;
    }

})();
```

Now we will add the object we will return and start adding in some of the data that we need. The turtlesData variable that contained the JSON from our list controller will be copied over into this factory and we will create a new variable to hold the JSON for the quiz questions.

```
function DataService(){

    var dataObj = {
        turtlesData: turtlesData,
        quizQuestions: quizQuestions
    };

    return dataObj;
}
```

We can see that we have created some entries in the dataObj which reference variables that contain JSON that we have copied into the script at the bottom. This is then returned from the factory so that our controllers can use this data.

The turtlesData is the exact same JSON as we used before. The quizQuestions is new JSON which can be seen here:

```
var quizQuestions  = [
        {
            type: "text",
            text: "How much can a loggerhead weigh?",
            possibilities: [
                {
                    answer: "Up to 20kg"
                },
                {
                    answer: "Up to 115kg"
                },
                {
                    answer: "Up to 220kg"
                },
                {
                    answer: "Up to 500kg"
                }
            ],
            selected: null,
            correct: null
        },
        {
            type: "text",
            text: "What is the typical lifespan of a Green Sea Turtle?",
            possibilities: [
                {
                    answer: "150 years"
                },
```

Taking a closer look at the JSON for each quiz question we see that it has a type (text, or image) which allows us to have different styles of questions, the text of the question itself, four possible answers and two flags called selected and correct which are initialised to null. We will discuss all of this more later.

Angular Factories leads to Dependency Injection

We have seen it all before. So we will just go back to our controllers and add the “dataservice” factory to the array of dependencies. Also adding it as an argument to the controller functions.

Here is the quiz controller:

```
(function(){

    angular
        .module("turtleFacts")
        .controller("quizCtrl", QuizController);

    QuizController.$inject = ['quizMetrics', 'DataService'];

    function QuizController(quizMetrics, DataService){
        var vm = this;

        vm.quizMetrics = quizMetrics;
        vm.dataService = DataService;

    }

})();
```

We also do the same in the list controller. While we are in there we can delete the turtlesData variable that we now have in the factory (if you haven’t already removed it).

```
(function(){

    angular
        .module("turtleFacts")
        .controller("listCtrl", ListController);

    ListController.$inject = ['quizMetrics', 'DataService'];

    function ListController(quizMetrics, DataService){

        var vm = this;

        vm.quizMetrics = quizMetrics;
        vm.data = DataService.turtlesData;
        vm.activeTurtle = {};
        vm.changeActiveTurtle = changeActiveTurtle;
        vm.activateQuiz = activateQuiz;
        vm.search = "";

        function changeActiveTurtle(index){
            vm.activeTurtle = index;
        }

        function activateQuiz(){
            quizMetrics.changeState(true);
        }
    }

})();
```

Notice that we previously had a line **vm.data = turtlesData** but of course, the turtlesData variable doesn't exist in the controller, it is now inside the dataservice factory. So we change that line to **vm.data = DataService.turtlesData** .

Before we forget, we need to add the new dataservice factory script to the tags at the bottom of our HTML. The scripts area of the HTML now looks like this:

```
<script src="js/app.js"></script>
<script src="js/controllers/list.js"></script>
<script src="js/controllers/quiz.js"></script>
<script src="js/factories/quizMetrics.js"></script>
<script src="js/factories/dataservice.js"></script>
```

Chapter 10

Getting Stylish With Ng-Class

Moving swiftly on to the markup for the quiz page in our application. The first thing we will tackle is the progress area and the legend. If you can't remember what these sections look like, take a look in the video or check the application. We will introduce a new directive to style the progress buttons – ng-class. This will add styles based on conditions we specify.

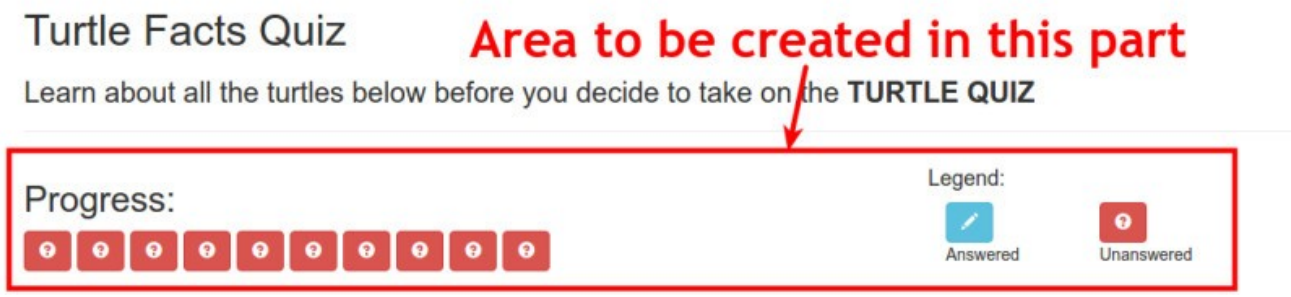
If you want to see the app for yourself, check it out here.

The git repo can be found here.

Click theimage to go to the video on youtube or go here https://www.youtube.com/watch?v=pVuOPOL-WoY&list=PLqr0oBkln1FBmOjK24_B4y_VAA8736wPq&index=10

The next part can be found here

Hopefully you have noticed, or realised that there is a button for every question in the quiz and they are all styled the same initially. This means we can use the ng-repeat directive to cut down on the code we need to write.



For us to have use ng-repeat to count the questions, we will need access to the questions in the view. We have already injected the data service into the controller. So no we just need to create a property that is attached to the view model.

```
function QuizController(quizMetrics, DataService){

    var vm = this;

    vm.quizMetrics = quizMetrics;
    vm.dataService = DataService;

}
```

Some Bootstrap HTML

We will contain the whole progress and legend area in a Bootstrap row. The progress bar will take up 8 columns and the legend the remaining 4. Fortunately, Bootstrap contains a nice class to style an area with a lot of buttons. It is called .btn-toolbar.

```
<div ng-controller="quizCtrl as quiz" ng-show="quiz.quizMetrics.quizActive">

    <div class="row">
        <div class="col-xs-8">
            <h2>Progress:</h2>
            <div class="btn-toolbar">

            </div>
        </div>

       // LEGEND AREA GOES HERE

    </div>

</div>
```

Inside the button toolbar we want a button for each of the questions in the quiz. These buttons will be styled to let the user know if they have answered that question or not and they will also allow the user to navigate between questions.

The ng-repeat directive will go onto the button markup and will loop for the total number of questions.

```
<div class="btn-toolbar">
      <button class="btn"
              ng-repeat="question in quiz.dataService.quizQuestions">

      </button>
</div>
```

In Walks The Ng-Class Directive

Ng-Class is a directive like any other. But this time, inside the quotes we put an object that contains name:value pairs. The name will be the class we want to add and the value will be the conditional expression or boolean that will determine whether or not that class should be added. We can add multiple name:value pairs if we want, each with their own class and condition.

```
<div class="btn-toolbar">
      <button class="btn"
              ng-repeat="question in quiz.dataService.quizQuestions"
              ng-class="{'btn-info': question.selected !== null, 'btn-danger': question.
selected === null}">

      </button>
</div>
```

Here, btn-info and btn-danger are the two respective classes that will be added and the conditions are if the selected flag of the current question is not null or is null respectively.

Glyphicons Also Join The Party

Now that we have the correct colouring to the buttons we need to add the icons. As you have see from the example, and answered question has the pencil icon while the unanswered has the question mark.

To do this will will use Glyphicons that come bundled with Bootstrap, which will be added to a span inside the button. Again, we will use ng-class to add either the glyphicon-pencil class or the glyphicon-question-sign class depending on if the question is answered or not.

```
<div class="btn-toolbar">
      <button class="btn"
              ng-repeat="question in quiz.dataService.quizQuestions"
              ng-class="{'btn-info': question.selected !== null, 'btn-danger': questio
n.selected === null}">

                  <span class="glyphicon"
                        ng-class="{'glyphicon-pencil': question.selected !== null, 'gly
phicon-question-sign': question.selected === null}"></span>

      </button>
</div>
```

Now Here Comes The Legend

Inside the same row we created for earlier we will add the legend area. This area doesn't have any fancy directives as it doesn't need to change throughout the lifetime of the application. It can be hard coded.

We will make the area the remaining four columns. The markup isn't complex so I won't waste your time by explaining it. I will just show you it.

```
<div class="col-xs-4">
    <div class="row">
        <h4>Legend:</h4>
        <div class="col-sm-4">
            <button class="btn btn-info">
                <span class="glyphicon glyphicon-pencil"></span>
            </button>
            <p>Answered</p>
        </div>
        <div class="col-sm-4">
            <button class="btn btn-danger">
                <span class="glyphicon glyphicon-question-sign"></span>
            </button>
            <p>Unanswered</p>
        </div>
    </div>
</div>
```

This markup goes directly below the markup for the progress area but still inside the row. Refer to the first HTML snippet on this page above, there is a comment showing where the legend markup goes.

Chapter 11

Bootstrapping The Quiz Questions

No waiting around here, let's just jump straight into creating the markup for the questions in the quiz. The whole area will be surrounded in a bootstrap well and we will pull the questions from the data service we created before.

If you want to see the app for yourself, check it out here.

The git repo can be found here.

Click the image to go to the video on youtube or go here https://www.youtube.com/watch?v=UC8aWbE9_G4&index=11&list=PLqr0oBkln1FBmOjK24_B4y_VAA8736wPq

Here is the next part

The base of the HTML shouldn't be new to you, so I will just show you the code with the bootstrap well and the foundation of the question markup.

```
<div class="row">
    <h3>Question:</h3>
    <div class="well well-sm">
        <div class="row">
            <div class="col-xs-12">
                <!-- Question Area -->

                <h4><!-- The question will go in here --></h4>

            </div>
        </div>
    </div>
</div>
```

Inside this div is where we will need to add the text for the question as well as all possible answers for that question. We could just ng-repeat all questions, but they would then display all the questions in a column down the page. We want only one question with a continue button to take us to the next (or to use the buttons in the progress section).

So instead, what we will do is to create a property on our controller that indicates the question that is active. Then the corresponding text and possible answers will be displayed for that question. Doing it this way allows us to easily move to the next question by incrementing the active question.

Active Question

First, we need to create the activeQuestion property inside the quiz controller. We will initialise this to 0. This is because we are starting from the first question, which of course, in programming languages tends to start from a 0 index.

This is what our quiz controller looks like thus far.

```
(function(){

    angular
        .module("turtleFacts")
        .controller("quizCtrl", QuizController);

    QuizController.$inject = ['quizMetrics', 'DataService'];

    function QuizController(quizMetrics, DataService){

        var vm = this;

        vm.quizMetrics = quizMetrics;
        vm.dataService = DataService;
        vm.activeQuestion = 0;

    }

})();
```

To make things look a bit nicer, I like to add the question number before the question. Something like this:

1. This is a wonderful question that you will know the answer to.

We could just start with the number one and increment the number as the user progresses through the quiz. The problem there is that that wouldn't allow the user to go back to a question, which is a feature we want in our quiz.

But we have the activeQuestion property which holds a number of the current question, just in 0 index form. So all we need to do is to add 1 to that value. Then we can print out the text of the question which we get from the question JSON in the dataservice by accessing the index of the active question.

```
<h4>{{quiz.activeQuestion+1 + ". " +
quiz.dataService.quizQuestions[quiz.activeQuestion].text}}</h4>
```

Now We Bring Out The Ng-Repeat Gunz

Directly below the h4 we just created we will create a new row that will house the possible answers. Each answer will take up half the width of the well, so we will give it the corresponding bootstrap class.

On the div for the question, we can use ng-repeat to repeat itself for all four possible answers to the question. This is the code that we will add right after the h4 from the last few paragraphs.

```
<div class="row">
    <div class="col-sm-6" ng-repeat="answer in quiz.dataService.quizQuestions[quiz.activ
eQuestion].possibilities">
        <h4 class="answer">
            <!-- Possible answers go here -->
        </h4>
    </div>
</div>
```

So here we are referencing the possible answers using a long string similar to the one we created to get the question text, except this time we are referencing `.possibilities` instead of `.text` . But we are still getting that by finding the activeQuestion index of the quizQuestions property on the dataService which we have access to in the quiz controller.

The ng-repeat has the alias of answer and the possibilities object that we are looping through each have a value of answer, which is the text of the possible answer. Just to jog your memory, here is a snippet of the JSON we are dealing with.

```
{
    type: "text",
    text: "What is the typical lifespan of a Green Sea Turtle?",
    possibilities: [
        {
            answer: "150 years"
        },
        {
            answer: "10 years"
        },
        {
            answer: "80 years"
        },
        {
            answer: "40 years"
        }
    ],
    selected: null,
    correct: null
}
```

Now using the ng-repeat alias of answer (which will be looping through the object of possibilities) we can reference the answer value in each object. This gives us answer.answer. We will add this like so:

```
<div class="row">
    <div class="col-sm-6" ng-repeat="answer in quiz.dataService.quizQuestions[quiz.activ
eQuestion].possibilities">
        <h4 class="answer">
            {{answer.answer}}
        </h4>
    </div>
</div>
```

Adding Custom Styling To The Questions

The answers may be there right now, but they are certainly not looking pretty like we want them to. So we need to add our own custom styling.

Adding the class of answer to the possible answers h4 will then allow us to add the following to the style.css

```
.answer{
    padding: 15px 20px;
    border-radius: 10px;
    border: 1px solid #bbb;
}
.answer:hover{
    cursor: pointer;
}
```

Creating The Continue Button

This will be a simple case of adding a button element to the HTML just at the bottom of the current bootstrap well. The button will be styled using some bootstrap classes and we will add an ng-click directive to trigger the functionality that will allow us to change the active question (which will in turn, automatically change the text and answers).

```
<button class="btn btn-warning" ng-click="quiz.questionAnswered()">Continue</button>
```

The function that we reference here is one we have not created yet but that is what we will tackle in the next part of this series.

Chapter 12

Controller Logic Keeps Us Going

We set ourselves up nicely in the last part. The ng-click calls a function on the quizController every time the “continue” button is clicked. This function will then need to change the activeQuestion property to the next available unanswered question in the quiz. So follow along with the article or the video as we build this controller logic.

If you want to see the app for yourself, check it out here.

The git repo can be found here.

Video Tutorial, For The More Visually Inclined

Click the image to go to the video on youtube or go here https://www.youtube.com/watch?v=XamHS-0PiMM&list=PLqr0oBkln1FBmOjK24_B4y_VAA8736wPq&index=12

The next part can be found here.

Let’s get started on the questionAnswered function to get this functionality working. So let’s first take an overview look at what this function will need to do.

For starters, we need to check if the question the user has just clicked continue on has been answered or if it has been skipped. If it has been answered we need to increment a variable that will count the total number of question that have been answered in the quiz so far. This will be used to allow us to know when the user has finished. We will obviously need to create this variable on the controller.

Once we have incremented the total questions answered, we check if the quiz has been finished ie. the total questions answered is equal to the total number of questions in the quiz.

If the quiz isn’t finished then we must increment the activeQuestion property to the next available unanswered question.

Controller Logic To Check If The Question Has Been Answered

If you remember back to the JSON for each question you will remember that there is a selected flag for each question. This is initially set to null, meaning the question has not been answered. As soon as the user does answer the question then this selected property will be set to the index of the possible answer they selected (0-3).

We still need to create the function that will set this selected value, but just keep in mind that this is something we will be doing in the future.

With this knowledge, it is now possible to detect if the question has been answered by checking if the selected property for this question equals null. If it does, the question hasn't been answered and if it does then, of course, it has been answered.

This is the start of the questionAnswered function:

```
var numQuestionsAnswered = 0;

function questionAnswered(){

    if(DataService.quizQuestions[vm.activeQuestion].selected !== null){
        numQuestionsAnswered++;
    }

}
```

You can see that we initialise a numQuestionsAnswered variable. Notice that we used var and not vm. To create the variable. This is because the view does not need access to this variable. It is exclusively for the controller to use.

Then we check to see if the question has been answered, ie the selected property for this question doesn't equal null. If the question has been answered, then we increment the numQuestionsAnswered variable.

Check If All Questions In The Quiz Have Been Answered

Inside the conditional statement we incremented the number of questions answered. Now that we have done that, we don't want to move straight onto the rest of the function. Instead, we want to make sure the user hasn't just finished the quiz. There is no point doing all this extra logic if they have just answered the final question.

Still inside the conditional we can add a further conditional to check if we are at the end of the quiz. To check this we simply compare the number of answered questions to the total number of questions in the quiz.

```
function questionAnswered(){

    var quizLength = DataService.quizQuestions.length;

    if(DataService.quizQuestions[vm.activeQuestion].selected !== null){
        numQuestionsAnswered++;

        if(numQuestionsAnswered >= quizLength){
           //Finalise the quiz
        }
    }

}
```

Just to make things easier we stored the length of the quiz in a variable before the conditional, incase we need it again. The logic inside the second conditional is something we will return to later. But the logic will do some final checks and then return from the function call.

Now that the checks have been done the further logic that needs to be done when the quiz isn't finished can be added. At the end of the questionAnswered function, after the conditional blocks we can think about incrementing the activeQuestion to the next available unanswered question.

Avoiding Annoyances In Applications

This bit of code in our app isn't as simple as just incrementing activeQuestion. This is because we have the buttons in the progress bar that will allow the user to hop between questions.

For example, let's say the users skips the first question without answering it. They are taken to question two. They then answer it and are taken to question three. At this point they remember the answer to question one, so they hit the button in the progress area to take them back to question one.

Hitting this button will change activeQuestion to the index of the first question, 0 (we will implement this logic soon). Now they answer the question and hit continue. If we only incremented the activeQuestion when the user hits continue, they will be taken to question two, which they have already answered.

This is an annoyance, so I would rather avoid it.

Calling Another Function

Instead we need to scan through all the questions and find the lowest indexed question that has yet to be answered. In the previous example this logic would skip question two because it has been answered and it would instead find that question three is the lowest index that hasn’t been answered yet.

This is a bit more complex, so I would urge you to separate the logic into another function that we then simply call from the current function.

This is exactly what I have done and I will call the function setActiveQuestion. So I will create the function inside the controller just like we have been doing in the past and then call it from within the questionAnswered function using vm.setActiveQuestion().

```
function QuizController(quizMetrics, DataService){

    var vm = this;

    vm.quizMetrics = quizMetrics;
    vm.dataService = DataService;
    vm.activeQuestion = 0;
    vm.questionAnswered = questionAnswered;
    vm.setActiveQuestion = setActiveQuestion;

    var numQuestionsAnswered = 0;

    function setActiveQuestion(){
        // Will create this logic shortly
    }

    function questionAnswered(){

        if(DataService.quizQuestions[vm.activeQuestion].selected !== null){
            numQuestionsAnswered++;
        }
        vm.setActiveQuestion();
    }
}
```

Here the setActiveQuestion function is obviously blank, but you can see how it was created and then called.

setActiveQuestion Is A Bit Loopy

Let’s now start building out the setActiveQuestion function. Here are the first few lines inside the function.

```
function setActiveQuestion(){
    var breakOut = false;
    var quizLength = DataService.quizQuestions.length - 1;

    while(!breakOut){
        // Indefinite Loop
    }
}
```

We are using a while loop here to loop indefinitely until we find a question that has yet to be answered. So we use a variable called breakOut, which is set to false and loop continuously, while breakOut is false. As soon as we find an unanswered question we will set breakOut to true, which will break the while loop and all is good.

The quizLength variable is pretty self explanatory as it is very similar to the one we created earlier. The main difference this time is that we are taking 1 away from the length. We do this because activeQuestion is 0 index. So if the length of the quiz is 10 questions, the activeQuestion will cycle from 0-9. Hence we take one away from the actual length to make it 0 index too.

Inside the while loop we will add a kind of fancy looking line of code, but it achieving a simple task.

```
while(!breakOut){
    vm.activeQuestion = vm.activeQuestion < quizLength?++vm.activeQuestion:0;
}
```

This is basically a condensed if statement. It checks if the activeQuestion is less than the length of the quiz. If it is, it simply increments activeQuestion and if activeQuestion isn't less than the length ie, it is the same size (last question in quiz) then we set activeQuestion back to 0 to continue looping from the start.

The reason we want this line of code to increment or reset activeQuestion to zero is simple. Say we skip question 1, but then answer all other questions. When we click continue on the final question, the code will check if all questions have been answered, which they havent been. This will then trigger the setActiveQuestion function.

If we only increment activeQuestion here, we will end up incrementing it past the total length of the quiz. Which would be invalid. So instead, we want to go back to the start of the quiz to find where that unanswered question is.

We know there is an unanswered question because we checked if total number of questions answered was equal to total questions and it wasn't. So we know there is an unanswered question. So go back and keep going until we find it.

Now we need some logic that will check if the current activeQuestion, after that last line (the incremented version or the reset back to zero) is unanswered.

```
while(!breakOut){
    vm.activeQuestion = vm.activeQuestion < quizLength?++vm.activeQuestion:0;


    if(DataService.quizQuestions[vm.activeQuestion].selected === null){
        breakOut = true;
    }
}
```

Hopefully, this conditional make a bit of sense to you. We are simply testing to see if the question is unanswered, if it isn’t then the while loop loops again and we again increment activeQuestion or reset it to zero if we where on the last question.

If the question is unanswered then the code inside the conditional runs and we set breakOut to true, which breaks the loop and ultimately returns the function with the newly found unanswered question as the active question.

Chapter 13

What (ng-)if You Don’t Want To Display It All

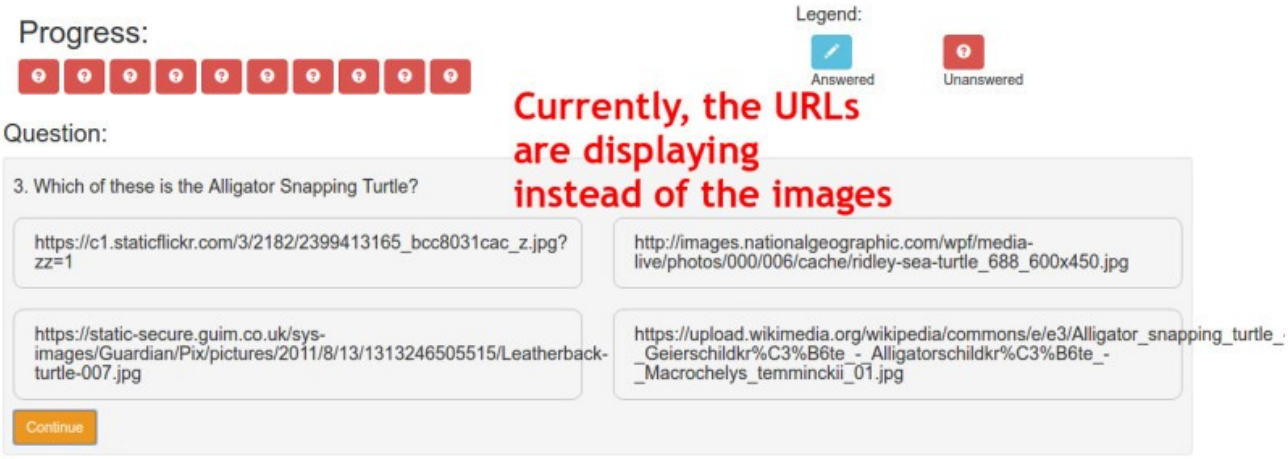
The quiz is starting to look distinctly like a quiz! Go us. But unfortunately, the image questions do not display correctly. They are showing the url instead of the image. In this part we will introduce the ng-if directive to solve that problem.

If you want to see the app for yourself, check it out here.

The git repo can be found here.

Click the image or go to https://www.youtube.com/watch?v=BRD5DUQgLF4&list=PLqr0oBkln1FBmOjK24_B4y_VAA8736wPq&index=13

The next part can be found here



The way we will solve this is by duplicating the html that displays the possible answers using ng-repeat. Then instead of displaying the text using answer.answer inside an h4, we will add an image using ng-src=”answer.answer”. This way the url is in the place it should be – the src attribute of an image.

```
<div class="row">
    <div class="col-sm-6" ng-repeat="answer in quiz.dataService.quizQuestions[quiz.acti
veQuestion].possibilities">
        <div class="image-answer">

            <img ng-src="{{answer.answer}}">

        </div>
    </div>
</div>
```

Note that we have surrounded the image with a div of class image-answer. This will be used later to make sure that all images are uniform size on the page and not displaying in their native sizes.

However, we now have two blocks of code that will both display. When it’s a text based question, the text will display nicely, but we will also have an image that is trying to fetch a url that doesn’t exist (the text of the possible answers is obviously not a url). We get the opposite problem when it’s an image question.

So what we want to do is somehow only show one of these code blocks depending on what style of question the currently active question is. This is where the ng-if directive comes in.

What Exactly Does This Ng-If Do?

Ng-If achieves the exact functionality we are looking for. It accepts an expression that evaluates to a boolean; if this value is true if will add the html that it is attached to, to the page. If the value is false the html will not be rendered at all.

This may sound similar to what the ng-show and ng-hide directives do, but there is one key distinction. Ng-show and ng-hide are just like adding **display: none** to the css for that element. The element is still there in the html document, it just cannot be seen on the screen – it is hidden. But ng-if does not even render the html to the page at all.

The reason we are using ng-if here instead of ng-show or hide is because if we use ng-show or hide we will still end up with images that have src attributes that are not urls. The browser will still try to find them as urls and when they can't they throw errors. Ng-if never allows this scenario and is thus a bit cleaner. We like clean code.

Here is the two blocks of code with the ng-if directives added to both

```
<div class="row"
     ng-if="quiz.dataService.quizQuestions[quiz.activeQuestion].type === 'text'">
         <div class="col-sm-6" ng-repeat="answer in quiz.dataService.quizQuestions[qui
z.activeQuestion].possibilities">
             <h4 class="answer">
                 {{answer.answer}}
             </h4>
         </div>
</div>

<div class="row"
     ng-if="quiz.dataService.quizQuestions[quiz.activeQuestion].type === 'image'">
         <div class="col-sm-6" ng-repeat="answer in quiz.dataService.quizQuestions[qui
z.activeQuestion].possibilities">
             <div class="image-answer">
                 <img ng-src="{{answer.answer}}">
             </div>
         </div>
</div>
```

As you can see, this is why we added the question type to the JSON for each question. We can now simply query what the type of the currently active question is and we are done.

MOAR CSS Rules!

There isn’t much css that needs to be added to fix the sizing issues that we are left with on the images. But they need to be done.

```
.image-answer{
    cursor: pointer;
    height: 350px;
    width: 100%;
    overflow: hidden;
    border-radius: 10px;
    margin-bottom: 20px;
}
.image-answer img{
    width: 100%;
    height: auto;
}
```

All of this should be pretty self explanatory. It will scale up/down all images to the same size and style them nicely with rounded corners etc. We also add the pointer cursor to indicate it is clickable to the user.

Chapter 14

Index For Ng-Repeat Will Help User Feedback

We now have the questions all displaying nicely in the quiz. The problem we face now is that when the user selects an answer there is no visual feedback to let them know that they have selected that answer. In this part we will introduce index for ng-repeat and create that feedback for the users.

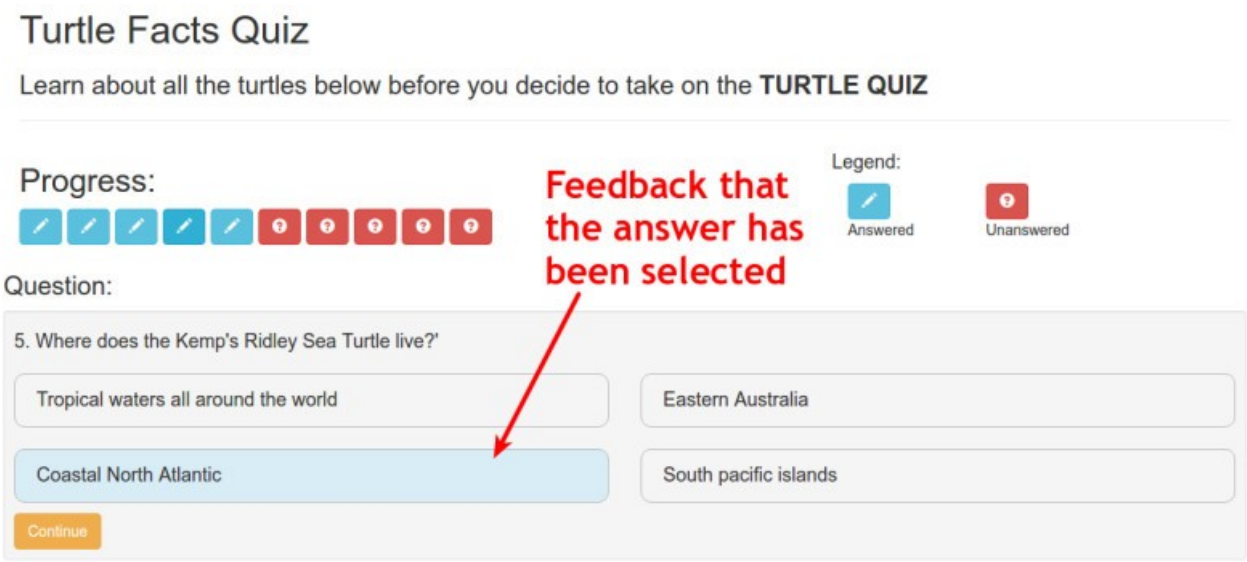
If you want to see the app for yourself, check it out here.

The git repo can be found here.

Click the image to go to the video on youtube or go here https://www.youtube.com/watch?v=lQ4lVuGC2F4&list=PLqr0oBkln1FBmOjK24_B4y_VAA8736wPq&index=14

The next part can be found here

Getting started on the text questions first we will go into the h4 that holds each possible answer. Here we will add an ng-class that will give it a blue (bg-info) background when the user selects that answer. Watch the video tutorial to see this in action.



But how do we use ng-class to only add the background when the user clicks on that answer? Well, we simply combine the efforts of ng-click and ng-class into one.

Remember that selected property on the JSON for each question? We can use ng-click to change that to the index of the question that was clicked on and use ng-class to display the background only if the index of that answer is the same as the index in the selected property for that question.

How Do We Tell The Ng-Click The Index Of What Was Clicked?

Fortunately, ng-repeat provides us with a handy parameter that can handle this for us – $index. This does exactly what is says on the tin. It will give us the index of the current iteration through the ng-repeat.

For example, on the first possible answer, $index = 0. On the second possible answer $index = 1 etc etc.

Bringing It All Together

We can now give ng-click a function that will change the value of the selected property for the current question. We will do this by passing $index in as an argument to the function in ng-click.

In the ng-class we can make a comparison between the $index of all the possible answers and the selected property for the question. Only the possible answer that matches will be given the background.

```
<div class="row"
     ng-if="quiz.dataService.quizQuestions[quiz.activeQuestion].type === 'text'">
         <div class="col-sm-6" ng-repeat="answer in quiz.dataService.quizQuestions[qui
z.activeQuestion].possibilities">
             <h4 class="answer"
                 ng-class="{'bg-info': $index === quiz.dataService.quizQuestions[quiz.a
ctiveQuestion].selected}"
                 ng-click="quiz.selectAnswer($index)">

                     {{answer.answer}}

             </h4>
         </div>
</div>
```

Now let’s create the function in the quiz controller. This process will be familiar to you by now. I have left out all the rest of the code that we have added to the controller previously just to make sure you can see exactly what we are doing here.

```
function selectAnswer(index){
    DataService.quizQuestions[vm.activeQuestion].selected = index;
}
```

A simple one liner. We just set the selected flag of the active question to the index that was passed into the function. Easy stuff.

Why Does The Button In The Progress Area Change?

Now that we have done this we can happily click a possible answer and get some nice feedback showing that we have selected that answer. But you may have noticed that the corresponding button in the progress area also changes when we make a selection. Why?

You may not remember but earlier on, when we were creating the progress buttons we gave it an ng-class that depended on the selected flag also. Here is the code to jog your memory.

```
<button class="btn"
        ng-repeat="question in quiz.dataService.quizQuestions"
        ng-class="{'btn-info': question.selected !== null, 'btn-danger': question.selec
ted === null}"
        ng-click="quiz.setActiveQuestion($index)">

            <span class="glyphicon"
                  ng-class="{'glyphicon-pencil': question.selected !== null, 'glyphicon
-question-sign': question.selected === null}"></span>
</button>
```

So when the selected flag for each question changes it simultaneously and automatically changes the background of the selected answer, the background of the progress button and the glyphicon of the button too. NICE!

Chapter 15

Custom CSS For Image Feedback + Code Reuse

Having nice user feedback is great, but move forward to the image questions and you will find that it all breaks. There is no feedback whatsoever for the image questions. In this part we will fix that problem. Then we will see how we can start reusing code we have already written to make the progress buttons skip to their corresponding question.

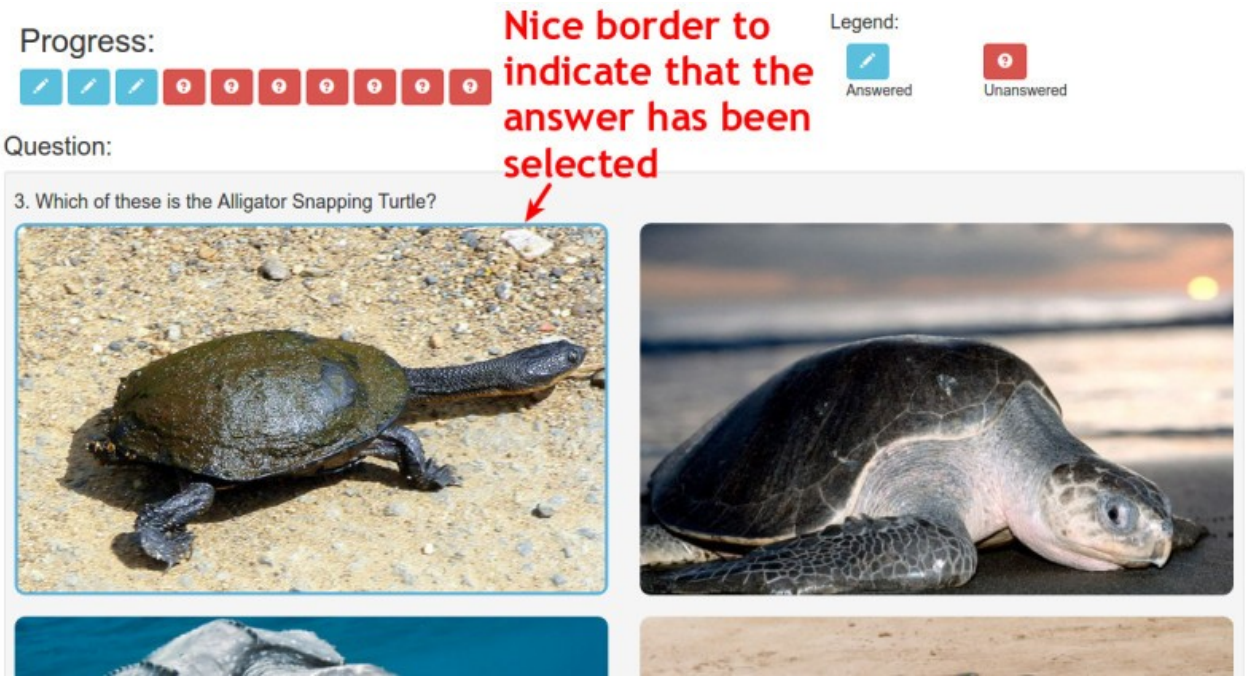
If you want to see the app for yourself, check it out here.

The git repo can be found here.

Click the image to go to the video on youtube or go here https://www.youtube.com/watch?v=kDQco9gfYmo&index=15&list=PLqr0oBkln1FBmOjK24_B4y_VAA8736wPq

The next part can be found here

Much like we did for the text questions, we will add an ng-class and ng-click directive onto the row that handles the images. Instead of giving it a bootstrap class though, we will give it a custom class that we will style ourselves. This is because a background would be useless for our image. Instead, we want a nice border.




```
<div class="row"
     ng-if="quiz.dataService.quizQuestions[quiz.activeQuestion].type === 'image'">

         <div class="col-sm-6" ng-repeat="answer in quiz.dataService.quizQuestions[qui
z.activeQuestion].possibilities">
             <div class="image-answer"
                  ng-class="{'image-selected': $index === quiz.dataService.quizQuestion
s[quiz.activeQuestion].selected}"
                  ng-click="quiz.selectAnswer($index)">

                      <img ng-src="{{answer.answer}}">

             </div>
         </div>
</div>
```

The ng-click is identical this time as it was to the last. We are still using $index to update the selected flag on the data for the question.

Let's Get Stylish

The css required for this is extremely simple. All we need to do is add a border. We don't need to round the edges as that has already been done by bootstrap. We add this line to the style.css.

```
.image-selected{
    border: 3px solid #56afdc;
}
```

The Progress Buttons Need Attention

We have spoken about how the buttons in the progress area will be used to allow the user to navigate to specific questions in the quiz. Currently though, clicking on these buttons will not do anything. So let's fix that by adding the functionality required.

Reusing Code Is Good Right?

Remember we have already created a function called setActiveQuestion that will increment the active question to the next available unanswered question. But what if we allowed the

function to accept an optional argument of the index of question to go to.

This when the function is given an question index as an argument it will just automatically set activeQuestion to that index. But if no argument is given then it will just find the next unanswered question like it is doing now. This sounds like a good plan to me.

First things first, go to the markup for the buttons in the progress area. On this button html, add and ng-click directive that triggers the setActiveQuestion function. We can take advantage of the fact we are using ng-repeat to create all the buttons and use $index to find the index of the current button. This is what we will pass as the argument to setActiveQuestion.

```
<button class="btn"
        ng-repeat="question in quiz.dataService.quizQuestions"
        ng-class="{'btn-info': question.selected !== null, 'btn-danger': question.selec
ted === null}"
        ng-click="quiz.setActiveQuestion($index)">

    <span class="glyphicon"
          ng-class="{'glyphicon-pencil': question.selected !== null, 'glyphicon-questio
n-sign': question.selected === null}"></span>
</button>
```

Time to add the new functionality into the setActiveQuestion function.

We start by using a conditional to check if an argument was passed into the function. If it wasn’t we run the code we already have and if it was we set active question to the index passed in. Simples.

```
function setActiveQuestion(index){

    if(index === undefined){
        var breakOut = false;

        var quizLength = DataService.quizQuestions.length - 1;

        while(!breakOut){

            if(DataService.quizQuestions[vm.activeQuestion].selected === null){
                breakOut = true;
            }

        }
    }else{
        vm.activeQuestion = index;
    }

}
```

Using **index === undefined** we can check if the function has been given an argument or not. Giving us the exact functionality we want. The else block here is also pretty self explanatory.

Chapter 16

Errors Must Be Handled In All Apps

Handling errors is a critical part of the life of a software developer. If you cannot handle errors, you will create code that has a ton of bugs and just doesn't function satisfactorily. In this part of the series we will use bootstrap alerts to display an error message when the user tries to continue at the last question but they haven't answered all questions yet.

If you want to see the app for yourself, check it out here.

The git repo can be found here.

Click the image to go to the video on youtube or go here https://www.youtube.com/watch?v=doU9zxZBoXI&list=PLqr0oBkln1FBmOjK24_B4y_VAA8736wPq&index=16

The next part can be found here

As I mentioned, the error should display when the user is on the last question and continues. If you remember back to the code in the setActiveQuestion function, we have some logic that increments the active question unless it is on the last question, in which case it resets it to 0.

When it resets to 0, it means the user was on the last question and clicked continue without all questions being answered (if all questions had been answered, the setActiveQuestion function would never have been called). This is when we want to display the error.

Using this knowledge of the reset back to 0, we can create a conditional after the potential reset to 0 that checks if it has infact been reset. If it has, then we display the error.

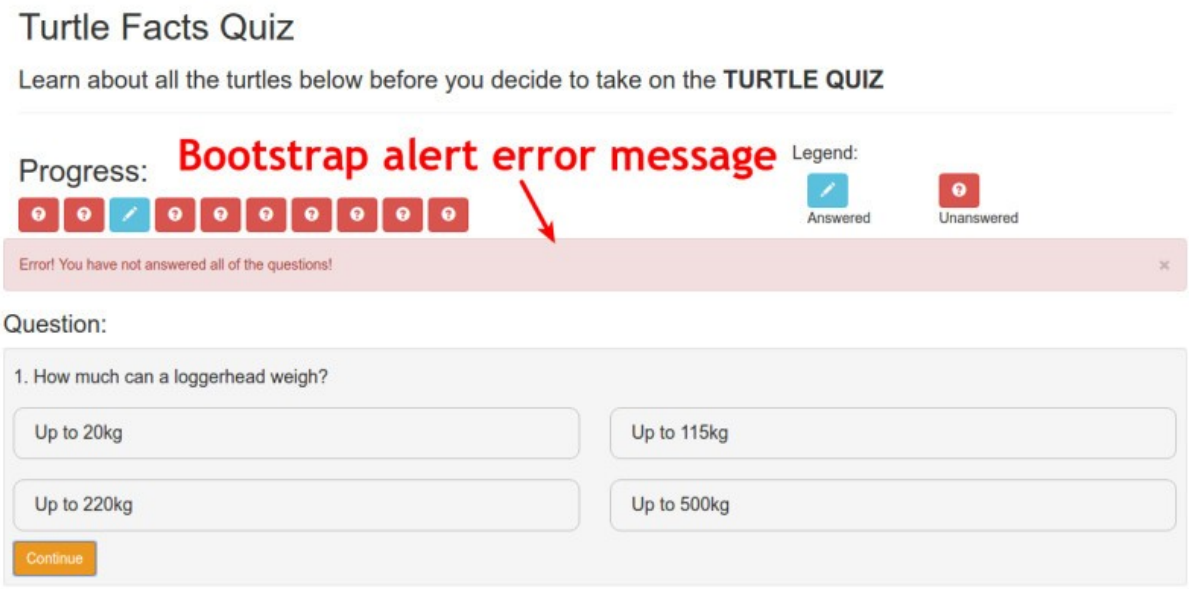
```
while(!breakOut){

    vm.activeQuestion = vm.activeQuestion < quizLength?++vm.activeQuestion:0;

    if(vm.activeQuestion === 0){
        vm.error = true;
    }

    if(DataService.quizQuestions[vm.activeQuestion].selected === null){
        breakOut = true;
    }
}
```

Notice inside the new conditional we reference `vm.error`. We have yet to create this, so add it at the top of the controller where we declare all the other properties. This what the whole quiz controller looks like so far:

```
(function(){

    angular
        .module("turtleFacts")
        .controller("quizCtrl", QuizController);

    QuizController.$inject = ['quizMetrics', 'DataService'];

    function QuizController(quizMetrics, DataService){

        var vm = this;

        vm.quizMetrics = quizMetrics;
        vm.dataService = DataService;
        vm.questionAnswered = questionAnswered;
        vm.setActiveQuestion = setActiveQuestion;
        vm.selectAnswer = selectAnswer;
        vm.activeQuestion = 0;
        vm.error = false;

        var numQuestionsAnswered = 0;

        function setActiveQuestion(index){

            if(index === undefined){
                var breakOut = false;

                var quizLength = DataService.quizQuestions.length - 1;

                while(!breakOut){

                    vm.activeQuestion = vm.activeQuestion < quizLength?++vm.activeQues
tion:0;

                    if(vm.activeQuestion === 0){
                        vm.error = true;
                    }

                    if(DataService.quizQuestions[vm.activeQuestion].selected === null)
{
                        breakOut = true;
                    }
                }
            }else{
                vm.activeQuestion = index;
            }

        }

        function questionAnswered(){

            var quizLength = DataService.quizQuestions.length;

            if(DataService.quizQuestions[vm.activeQuestion].selected !== null){
                numQuestionsAnswered++;
                if(numQuestionsAnswered >= quizLength){
                    // Finalise Quiz. Will tackle the code to put here later in the co
urse
                }
            }

            vm.setActiveQuestion();
        }

        function selectAnswer(index){
            DataService.quizQuestions[vm.activeQuestion].selected = index;
        }
    }

})();
```

Next, Create The Markup For The Bootstrap Alert

At the top of the row that holds the question area will be where we place the bootstrap alert. Just above the <h3>Question:</h3>.



The markup to create the alert box is simple. We just need to add the class of alert and the class of alert-danger to make it red. I also added a button with the class of close and the value of `&times` which will create a nice x to close the alert box.

On this button is an ng-click directive that will change the error property back to false which will remove the error box when we click the x button. Instead of calling a function from the ng-click like we have been doing throughout this course, it is also possible to just make an assignment declaration.

Obviously, we only want this alert box to display when the error property is true (then disappear when we hit the close button that sets error back to false). To do this we can use ng-show.

```
<div class="alert alert-danger"
     ng-show="quiz.error">

         Error! You have not answered all of the questions!

         <button class="close" ng-click="quiz.error = false">&times</button>

</div>
```

This works because quiz.error is already a boolean value, so as it changes it will show and hide the error box.

Finalising The Quiz

You may remember earlier, inside the questionAnswered function we checked if the current question had been answered, then we checked if all questions had been answered. Inside the second conditional we said we would add code to finalise the quiz. This is what we need to do now. Here is the code to help your memory:

You may be wondering why we have the line **numQuestionsAnswered >= quizLength** rather than simply **numQuestionsAnswered === quizLength.** I mean, if all questions have been answered numQuestionsAnswered will be exactly equal to the length of the quiz. Right?

The problem is that we do not stop the user from clicking continue on the same answered question twice. For example, the user answers question 1 and clicks continue which will call the questionAnswered function and increment numQuestionsAnswered and move us to question two.

Now suppose that the user realises the answer they gave to question one was wrong. So they go back to question one, change their answer and click continue again. This will then call the questionAnswered function and increment numQuestionsAnswered.

This leaves us in a position where numQuestionsAnswered is 2 but the actual number of questions that have been answered is only 1. This is why we need to have a >= check.

This then leads us to a situation where the user may have numQuestionsAnswered greater than or equal to the length of the quiz but they still haven't answered all the questions. So we need to add a final sanity check that loops through all the questions just to make sure they are all answered.

```
if(numQuestionsAnswered >= quizLength){

    for(var i = 0; i < quizLength; i++){

        if(DataService.quizQuestions[i].selected === null){
            setActiveQuestion(i);
            return;
        }
    }

    // More code here shortly

}
```

We call setActiveQuestion and pass in i as an argument. We do this because we know that at index i, the question isn't answered, because the if statement triggered. So we don't need to waste computations trying to find the unanswered question, we can just pass it in.

When setActiveQuestion is finished, we return from the function. This is because if we didnt return, the function would keep running and get down to the line below that again calls setActiveQuestion. There is no need for this, so we just return from the function.

All Checks Passed, Time To End The Quiz

If all questions have indeed been answered, then the if statement will never trigger and we won't call setActiveQuestion or return from the function, the code below the for loop will start to run. So this is where we add the code to do the housekeeping for the quiz before we move onto the results.

```
if(numQuestionsAnswered >= quizLength){

    for(var i = 0; i < quizLength; i++){

        if(DataService.quizQuestions[i].selected === null){
            setActiveQuestion(i);
            return;
        }
    }

    vm.error = false;
    vm.finalise = true;
    return;

}
```

The finalise property also needs to be initialised at the top of the controller, so don't forget to do that.

Using the finalise property that is set to true when all questions have been answered we can display a little prompt that asks the user if they are sure they want to continue to the results, or if they would like to stay on the quiz to change some answers.

Chapter 17

The End Of The Quiz Controller Is Near

The only thing left for us to do with the quiz controller is just prompt the user when they have finished just to confirm they want to move onto the results page. The finalise flag will come in useful here to allow the use of ng-show to show the prompt when the end is reached.

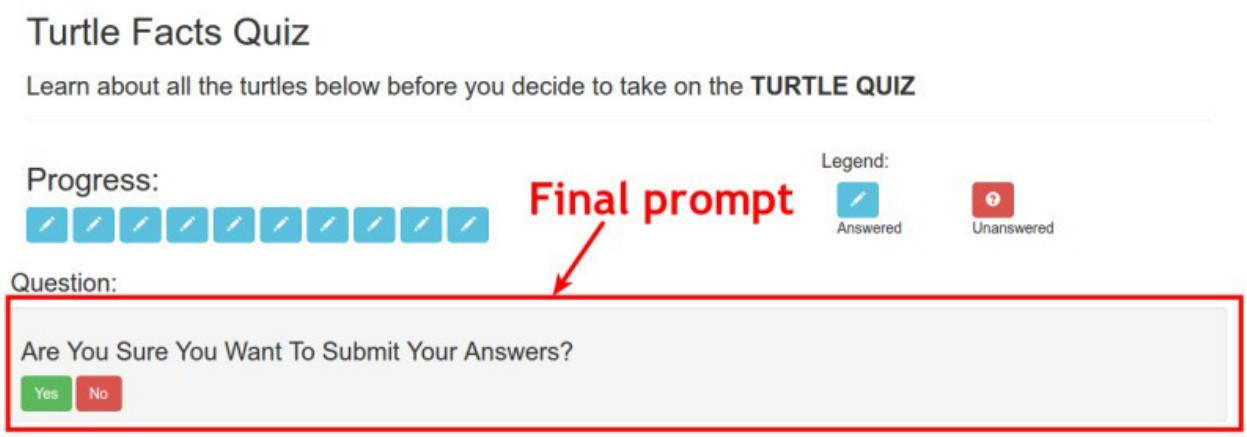
If you want to see the app for yourself, check it out here.

The git repo can be found here.

Click on the image to go to the video on youtube or go here https://www.youtube.com/watch?v=6uZfUfB4bN8&list=PLqr0oBkln1FBmOjK24_B4y_VAA8736wPq&index=17

The next part can be found here

The code for the little prompt is simple, it is just a bootstrap well with two buttons. We will place this markup at the end of the row that contains the well for the questions. This is the markup we will need.

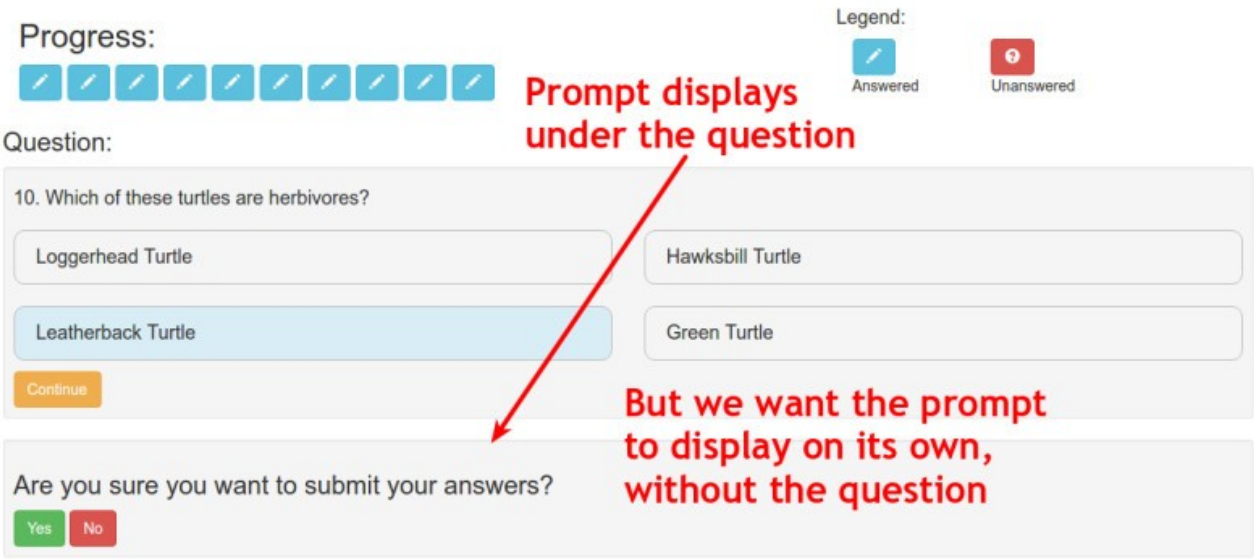



```
<div class="well well-sm" ng-show="quiz.finalise">
    <div class="row">
        <div class="col-xs-12">
            <h3>Are you sure you want to submit your answers?</h3>
            <button class="btn btn-success" ng-click="quiz.finaliseAnswers()">Yes</butt
on>
            <button class="btn btn-danger" ng-click="quiz.finalise = false">No</button>
        </div>
    </div>
</div>
```

Onto the well that of this markup we have added the trusty ng-show directive that will only show this prompt when the finalise property is true.

Each of the buttons both have an ng-click directive attached onto them. The no button simply sets finalise back to false which removes the prompt and keeps the user in the quiz to make some changes. Simple enough.

The yes button contains a function we will need to create. This function will reset all the properties and variables that we have been using throughout the course of the life of the quiz.



You may have noticed that the prompt simply displays under the quiz questions at the minute. This isn't exactly what we want. Instead, we want the questions to hide while the prompt is showing just to avoid possible confusion. This way the user is in no doubt what they have to do. We do this by adding an ng-hide to the question well div.

```
<h3>Question:</h3>
<div class="well well-sm" ng-hide="quiz.finalise">
```

Another Method Inside The Controller

The final thing for us to do is create the function that will run when the user clicks the yes button in the final prompt – the finaliseAnswers function. I won't bore you by repeating the process to add another function to the controller.

```
function finaliseAnswers(){

    vm.finalise = false;
    numQuestionsAnswered = 0;
    vm.activeQuestion = 0;
    quizMetrics.markQuiz();

}
```

Note that this is just the function itself. This will be inside the quiz controller and it will be added to the view model in the same way as before – using a named function – something like `vm.finaliseAnswers = finaliseAnswers.`

The first three lines inside the function are pretty self explanatory – just resetting the properties to their default values.

Next, we will call a function which we will create in the next part. This function will mark the answers the user gave against the correct answers to the quiz.

Refactoring the changeState Function

Earlier we created a function called changeState that takes a boolean argument and changes the state of the quiz to that boolean. However, shortly we will be creating another controller – the results controller. This means the results controller will also have a state that will need manipulating.

So let’s modify the changeState function to take a second argument which will be the metric to change – quiz or results controller. This way the same function can be used to manipulate the state of both areas of the quiz.

Here is the new version of the changeState function.

```
function changeState(metric, state){
    if(metric === "quiz"){
        quizObj.quizActive = state;
    }else if(metric === "results"){
        quizObj.resultsActive = state;
    }else{
        return false;
    }
}
```

I also added a new property to the quizObj object to make this possible – resultsActive. This is needed to ensure we actually have a state to change.

```
var quizObj = {

    quizActive: false,
    resultsActive: false,
    changeState: changeState

};
```

With this new function, we can complete the finaliseAnswers function by setting the quiz state to false (which removes the quiz from the view) and set the results state to true, which will take us to the results page.

```
function finaliseAnswers(){

            vm.finalise = false;
            numQuestionsAnswered = 0;
            vm.activeQuestion = 0;
            quizMetrics.markQuiz();
            quizMetrics.changeState("quiz", false);
            quizMetrics.changeState("results", true);

}
```

Remember To Go Back And Change What We Have Broken

Right now, we have broken the call to the changeState function we had in the list controller that triggered the quiz in the first place. We only passed in a state and not a metric to that. The code we broke was in the activateQuiz function inside the list controller.

```
function activateQuiz(){

    quizMetrics.changeState("quiz", true);

}
```

Now the code is working again.

Chapter 18

No Quiz Is Complete Without It Being Marked

In the last part we reference a function on the quizMetrics factory that we have not yet created. So it will be the task of this part to create that function – a function that will mark the answers to the quiz and calculate how many correct answers that user gave. Let's get on with marking the quiz.

If you want to see the app for yourself, check it out here.

The git repo can be found here.

Click the image to go to the video on youtube or go here https://www.youtube.com/watch?v=mBDbwKr4DyQ&list=PLqr0oBkln1FBmOjK24_B4y_VAA8736wPq&index=18

The next part can be found here

Without wasting any time, jump into the quizMetrics factory and get started. The first thing we will need to mark the quiz is the correct answers to all the questions. The correct answers will be another piece of data in the dataService factory. Much like we have copy and pasted in the data for the list and the quiz, we will also paste in the data for the correct answers.

So at the bottom of the data service factory, paste the following:

```
var correctAnswers = [1, 2, 3, 0, 2, 0, 3, 2, 0, 3];
```

Then inside dataObj of that service we add the correctAnswers. Like this:

```
var dataObj = {
    turtlesData: turtlesData,
    quizQuestions: quizQuestions,
    correctAnswers: correctAnswers
};
```

To get hold of this data from inside the quizMetrics factory we will need to inject the dataService into the factory (yes, we can inject dependencies into services too!). This process is exactly the same as the dependency injection that you have seen already in the controllers.

```
angular
    .module("turtleFacts")
    .factory("quizMetrics", QuizMetrics);

    QuizMetrics.$inject = ['DataService'];

    function QuizMetrics(DataService){
        // Rest of quizMetrics code left out to keep snippet clean
    }
```

The Function That Will Be Marking The Quiz

Now that we have the correct answers and have access to them in the quizMetrics factory, we can create the markQuiz function. Before we start that we will add a reference to that function in the dataObj of the quizMetrics factory, along with a few properties.

These other properties include a property to track the total number of correct answers the user has given (defaulted to 0), and a property to house the correct answers inside the quizMetrics factory object. Of course, we could just access the answers from the dataService, but the answers are a bit of data related to the quiz, so we will just keep things easy to understand and keep that stored there.

```
var quizObj = {
    quizActive: false,
    resultsActive: false,
    changeState: changeState,
    correctAnswers: [],
    markQuiz: markQuiz,
    numCorrect: 0
};
```

Now onto the implementation of the markQuiz function. This is simply a case of looping through all the questions in the quiz and then comparing the answer the user gave (which is stored in the selected property for each question) against the correct answer. If they match, we will set the “correct” property in the data to true and increment the numCorrect property. If the answers do not match then the correct flag is set to false.

```
function markQuiz(){

    quizObj.correctAnswers = DataService.correctAnswers;

    for(var i = 0; i < DataService.quizQuestions.length; i++){
        if(DataService.quizQuestions[i].selected === DataService.correctAnswers[i]){
            DataService.quizQuestions[i].correct = true;
            quizObj.numCorrect++;
        }else{
            DataService.quizQuestions[i].correct = false;
        }
    }

}
```

Chapter 19

First Steps To Getting The Results

We have finished the quiz controller so now it is time to start the results controller. In the last part we created a property called resultsActive, which is what we will use to trigger the results area to show using an ng-show. We will also delve into some more angular dependency injection.

If you want to see the app for yourself, check it out here.

The git repo can be found here.

Click the image to go to the video on youtube or go here https://www.youtube.com/watch?v=N6N3KhkKk3o&list=PLqr0oBkln1FBmOjK24_B4y_VAA8736wPq&index=19

The next part can be found here

Before we can start showing the results area we need to create it. So in the HTML, under the quiz controller markup let's create a new div with a new ng-controller directive. Again using the "controller as" syntax.

```
<div ng-controller="resultsCtrl as results">
    <!-- results controller markup -->
</div>
```

Now the controller itself needs to be created. So create a new file called results.js inside the controllers directory. The script will be started with an IIFE as using and the rest should also look familiar to you by now.

```
(function(){

    angular
        .module("turtleFacts")
        .controller("resultsCtrl", ResultsController);

    function ResultsController(quizMetrics, DataService){
        var vm = this;
    }

})();
```

More Angular Dependency Injection

I mentioned that we will change the visibility of the results controller markup depending on the value of the resultsActive property on the quizMetrics factory object. This means that we need access to that factory from inside the results controller – this means dependency injection!

```
ResultsController.$inject = ['quizMetrics', 'DataService'];

function ResultsController(quizMetrics, DataService){
    // Code for results controller here
}
```

Also don’t forget to add the script tag to the bottom of the html page.

```
<!-- Our application scripts -->
<script src="js/app.js"></script>
<script src="js/controllers/list.js"></script>
<script src="js/controllers/quiz.js"></script>
<script src="js/controllers/results.js"></script>
<script src="js/factories/quizMetrics.js"></script>
<script src="js/factories/dataservice.js"></script>
```

With the quizMetrics injected into the results controller we can use the ng-show directive to display the results markup.

```
<div ng-controller="resultsCtrl as results" ng-show="results.quizMetrics.resultsActive">

</div>
```

A Bit Of A Problem

If you now run through the application you will see that when the user clicks yes on the prompt at the end of the quiz, the list controller shows again. This is because we only hide the list controller when quizActive is true. But when the yes button is clicked in that prompt is sets quizActive back to false and resultsActive to true. So the list controller shows with the results controller below it.

We need to amend this by changing the ng-hide on the list controller to hide when either the quiz controller or the results controller is active.

```
<div ng-controller="listCtrl as list" ng-hide="list.quizMetrics.quizActive || list.quizM
etrics.resultsActive">

</div>
```

Chapter 20

Some Really Familiar Bootstrap Markup

This lesson isn't introducing anything new. Infact, you will have seen much of this code before. What we are tackling is the area at the top of the results controller that shows what questions the user got right and wrong along with a legend. This is very much like the progress area in the quiz controller. Let's take a look at some familiar bootstrap code.

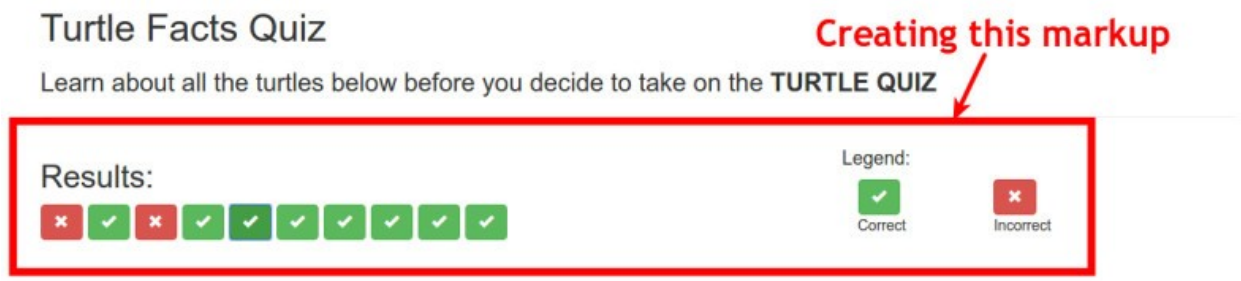
If you want to see the app for yourself, check it out here.

The git repo can be found here.

Click the image to go to the video on youtube or go here https://www.youtube.com/watch?v=AxQcLIRoxYw&index=20&list=PLqr0oBkln1FBmOjK24_B4y_VAA8736wPq

The next part can be found here

I say that it is similar, the markup is almost identical with a few key differences. Mainly the colors used are no longer red and blue but are instead red and green to indicate correct or incorrect. The glyphicons used are also different – a tick and an x this time.



Other than those differences the code is identical. So I will not bore you by explaining every line of the markup. Instead, I will just show you what we have.

```
<div class="row">
    <div class="col-xs-8">
        <h2>Results:</h2>
        <div class="btn-toolbar">

            <button class="btn"
                    ng-repeat="question in results.dataService.quizQuestions"
                    ng-class="{'btn-success': question.correct, 'btn-danger': !questio
n.correct}"
                    ng-click="results.setActiveQuestion($index)">

                        <span class="glyphicon"
                              ng-class="{'glyphicon-ok': question.correct, 'glyphicon-
remove': !question.correct}"></span>

            </button>
        </div>
    </div>
    <div class="col-xs-4">
        <div class="row">
            <h4>Legend:</h4>
            <div class="col-sm-4">
                <button class="btn btn-success">
                    <span class="glyphicon glyphicon-ok"></span>
                </button>
                <p>Correct</p>
            </div>
            <div class="col-sm-4">
                <button class="btn btn-danger">
                    <span class="glyphicon glyphicon-remove"></span>
                </button>
                <p>Incorrect</p>
            </div>
        </div>
    </div>
</div>
```

Note the key differences. We are now using the correct property we set to true or false on every question when we marked the quiz to style the buttons accordingly using ng-class on the button and the glyphicons.

We are referencing question.correct here has question is the alias for the ng-repeat. So each iteration through ng-repeat will be a different question in the quiz.

Also note that we do not have to say something like question.correct == true (although, this would also be valid and provide the output we need) because question.correct is already a boolean value so ng-class can evaluate it straight away. There is no need to add in extra computation when it is not needed.

Chapter 21

Deja Vu + A New Way To Use Ng-Class

Much like the previous part, the markup in this part will be very similar to that of the markup in the quiz controller. But we will spice things up a bit by adding new elements and showing a different way of using ng-class – using a function with ng-class instead of an object with name:value pairs.

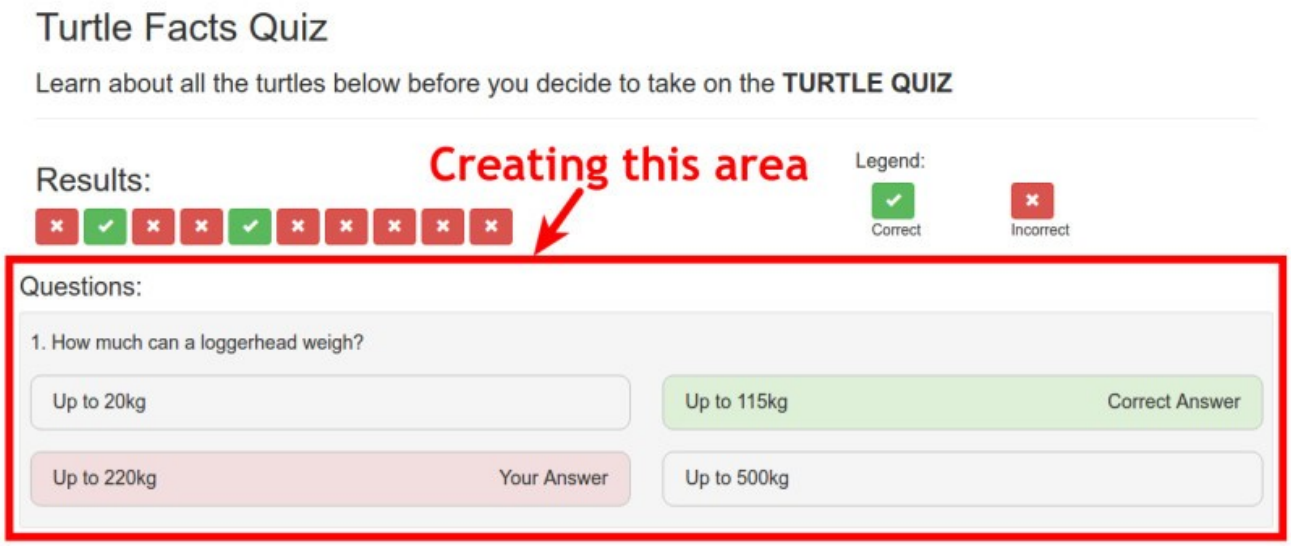
If you want to see the app for yourself, check it out here.

The git repo can be found here.

Click the image to go to the video on youtube or go here https://www.youtube.com/watch?v=bGyvJMprAcg&list=PLqr0oBkln1FBmOjK24_B4y_VAA8736wPq&index=21

The next part can be found here

Below the row we created in the last part we will continue to create the markup for the question area of the results section.




```
<div class="row">
    <h3>Questions:</h3>
    <div class="well well-sm">
        <div class="row">
            <div class="col-xs-12">

                <h4> {{}} </h4>

            </div>
        </div>
    </div>
</div>
```

Inside the {{}} syntax of the h4 we will need to reference the active question again to ensure the correct data is showing. This means we will need to create an active question variable on the results controller.

```
function ResultsController(quizMetrics, DataService){

    var vm = this;

    vm.quizMetrics = quizMetrics; // binding the object from factory to vm
    vm.dataService = DataService;

    vm.activeQuestion = 0;

}
```

Now that that's done we can write the code into that h4. It will be much like the code in the h4 of the quiz controller so if you don't quite understand what is going on here go back to the part where we created this markup in the quiz controller. Continuing on, we will also ng-repeat all the possible answers of the active question.

```
<div class="row">
    <h3>Questions:</h3>
    <div class="well well-sm">
        <div class="row">
            <div class="col-xs-12">

                <h4> {{results.activeQuestion+1 +". "+results.dataService.quizQuestions
[results.activeQuestion].text}} </h4>

                <div class="row"
                     ng-if="results.dataService.quizQuestions[results.activeQuestion].t
ype === 'text'">

                        <div class="col-sm-6" ng-repeat="answer in results.dataServic
e.quizQuestions[results.activeQuestion].possibilities">
                            <h4 class="answer">

                                    {{answer.answer}}

                            </h4>
                        </div>
                </div>

            </div>
        </div>
    </div>
</div>
```

Adding More Feedback To Each Answer

If you have seen the final application, you will notice that the answers in the results section display "correct answer" and if you gave a different answer it will display "your answer" on that on to let you know what answer you gave and what the actual correct answer was.

To implement that we need to create the two sets of text and ng-show then depending on the conditions of each possible answer.

```
<h4 class="answer">

    {{answer.answer}}

    <p class="pull-right"
       ng-show="$index !== results.quizMetrics.correctAnswers[results.activeQuestion] &
& $index === results.dataService.quizQuestions[results.activeQuestion].selected">Your A
nswer</p>

    <p class="pull-right"
       ng-show="$index === results.quizMetrics.correctAnswers[results.activeQuestion]">
Correct Answer</p>

</h4>
```

Fancy Expressions and a Function With Ng-Class

The expressions in the above code snippet may look a bit daunting initially but let’s walk through them.

Starting with the “Correct Answer” text. The expression in the ng-show is comparing the $index (which is the current index of the ng-repeat) to the correct answer for this particular question. We are doing this by referencing the correctAnswers property that we added to the quizMetrics factory earlier. Using activeQuestion to get the answer for the current active

question.

The ng-show on the “Your Answer” text is a bit more complicated. We are using two conditions here. The first one is checking the this possible answer is not the correct answer (using the same method as we did for the “Correct Answer” area). We do this as if the user gave the correct answer we can simply display “Correct Answer”, no need to tell them that was also the answer they gave. The absence of the “Your Answer” prompt implies their answer is correct.

The next condition to the expression is checking if this answer is in fact that answer that the user gave. We do this by referencing the selected property on the question, which of course stores the user’s answer for that particular question.

Both of these conditions have to be true, for the “Your Answer” text to display on the question.

The Questions Need Backgrounds

We want to style the correct answer with a green background and if the user answered incorrectly, style their answer with a red background. We will do this using ng-class. But instead of using the object notation for ng-class like we have done in the past, we will give it a function that returns the class to display.

```
<h4 class="answer"
    ng-class="results.getAnswerClass($index)">
```

It calls the getAnswerClass function (which we will create shortly) and passes in the $index from the ng-repeat. This function can then do some logic and figure out which class should be added and return that from the function. Whatever is returned by the function will be added as a class to the element by ng-class.

Heading into the results controller, let's create the function. All we will need to do inside the function is figure out if the argument passed in ($index) is the index of the correct answer. We will do this by referencing the array of correct answers we have used in the past.

If the argument is the correct answer, we return the green background class which is the bootstrap class of bg-success. If that conditional fails, there is an else if block that test if the argument passed in is the answer the user gave. If it is, then the class of bg-danger is returned, which is the red background.

```
vm.getAnswerClass = getAnswerClass;

function getAnswerClass(index){

    if(index === quizMetrics.correctAnswers[vm.activeQuestion]){
        return "bg-success";
    }else if(index === DataService.quizQuestions[vm.activeQuestion].selected){
        return "bg-danger";
    }

}
```

We can have the else if simply check if the answer is the one the user gave and not check if it also isn't the correct answer (like we did in the ng-show) because we already know the answer wasn't correct as the first if statement failed.

Chapter 22

Last Two Functions For The ResultsCtrl

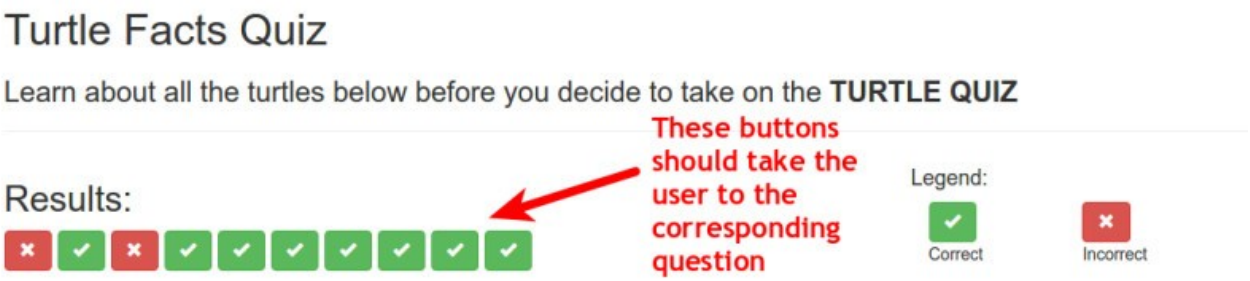
In this part we will add the final two functions into the results controller. A function that will be similar to the setActiveQuestion in the quiz controller but not nearly as complex and the second function will be to calculate the percentage score the user got in the quiz. We will also briefly discuss another filter – the Angular number filter.

If you want to see the app for yourself, check it out here.

The git repo can be found here.

Click the image to go to the video on youtube or go here https://www.youtube.com/watch?v=mWmfEtzGtRY&list=PLqr0oBkln1FBmOjK24_B4y_VAA8736wPq&index=22

The next part can be found here



The last thing we said in the last part was that clicking the buttons in the results area does nothing. So here we will create a function that allows the user to click any of those buttons and get taken to the corresponding question to see how they did.

So remember back to when we created the buttons in the results area, we added an ng-click that calls setActiveQuestion. But this of course, isn't the same as the setActiveQuestion on the quiz controller, it just has the same name. We now need to

create this new function.

In the ng-click we gave it an argument of $index, so in the function we will need to take that argument and set the active question to that index. Remember $index is coming from the ng-repeat. So if the user clicks on the 5th button from the left $index will equal 4 (0 index remember), then the function sets the active question to 4, so we see the corresponding data displaying.

```
vm.setActiveQuestion = setActiveQuestion;

function setActiveQuestion(index){

    vm.activeQuestion = index;

}
```

You may by now realise that we could get rid of this function and just add the code directly into the ng-click to change activeQuestion to $index. If you did think that, you are perfectly correct and in fact, that way is probably better. I only used the function to expose everyone to as many ways of doing things as possible to really get an idea of how Angular works.

Calculating User's Percentage Score

The final area to add to the results page is the bit in between the buttons and the question which shows the percentage score the user got. In the HTML, between the buttons and question we will add the following HTML markup.

```
<div class="row">
    <div class="col-xs-12 top-buffer">

        <h2>You Scored {{results.quizMetrics.numCorrect}} / {{results.dataService.quizQ
uestions.length}}</h2>

        <h2><strong>{{results.calculatePerc()}}%</strong></h2>

    </div>
</div>
```

Here we make use of the numCorrect property that we created earlier which is incremented in the markQuiz function everytime the user gets an answer correct. We also use the .length method on the quizQuestions object to find the total number of questions in the quiz.

The final thing you will notice is we call a function called calculatePerc. Which is what we will create now.

```
vm.calculatePerc = calculatePerc;

function calculatePerc(){

    return quizMetrics.numCorrect / DataService.quizQuestions.length * 100;

}
```

A Small Caveat – Angular Number Filter

In this quiz we have 10 questions which will result in the percentage always being a nice whole number. But if you have a different number of questions, you may end up with a percentage that has many decimal places. This can look bad and you should keep that under control.

There is another Angular Filter that will allow you to filter the number down to a certain number of decimal places. This would be how to do that:

```
<h2><strong>{{results.calculatePerc() | number:2}}%</strong></h2>
```

The standard | is used to tell Angular we are going to use a filter and the filter we use is called number. The a colon followed by the arguments for the filter. In this case, the only argument is the number of decimal places that you want to display.

Only Two Tasks Until We Have Finished The App

There are only two things left for us to do. The first is to fix the image questions again – just like we did in the quiz controller and the final task is to add the “go back to quiz” button. Then we are done.

Chapter 23

Our Old Enemy – The Image Questions

When we created the quiz controller we had the problem of the image urls displaying instead of the images themselves on image questions. We again face this problem in the results controller. Having already solved it once though, it should pose no problems to us now. We will use the Angular ng-if directive to fix the problem again.

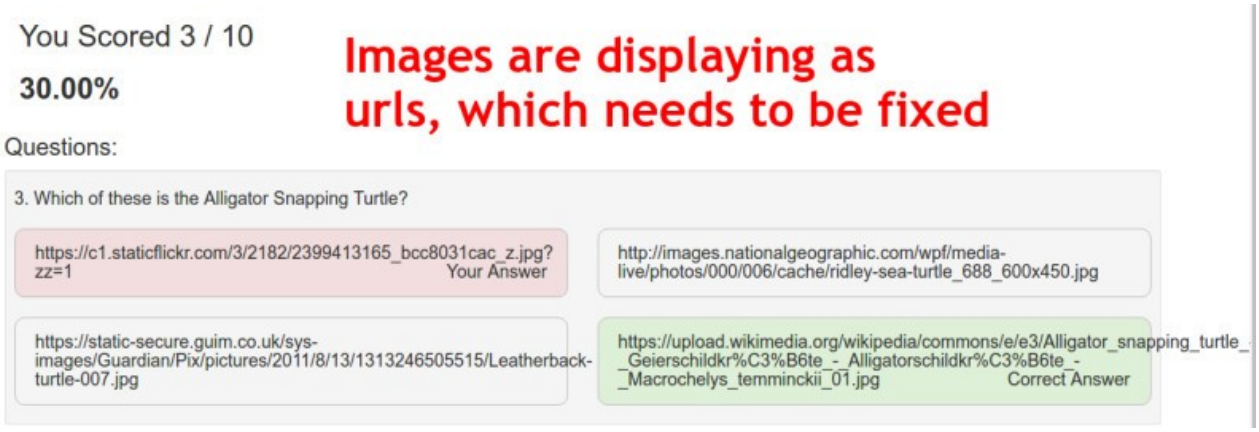
If you want to see the app for yourself, check it out here.

The git repo can be found here.

Click the image to go to the video on youtube or go here https://www.youtube.com/watch?v=xxH3dhPC5bY&list=PLqr0oBkln1FBmOjK24_B4y_VAA8736wPq&index=23

[The next part can be found here]

This time we want the images to display nicely but also want a border around the correct answer and if the user chose an incorrect answer, a border around the answer they selected. Both will provide feedback to the user.



Like we did in the quiz controller we just duplicate the row that contains the text questions and modify it slightly to house the image questions.

```
<div class="row">

        <div class="col-sm-6" ng-repeat="answer in results.dataService.quizQuestions[re
sults.activeQuestion].possibilities">

            <div class="image-answer"
                 ng-class="results.getAnswerClass($index)">
                    <img ng-src="{{answer.answer}}">
            </div>

    </div>
</div>
```

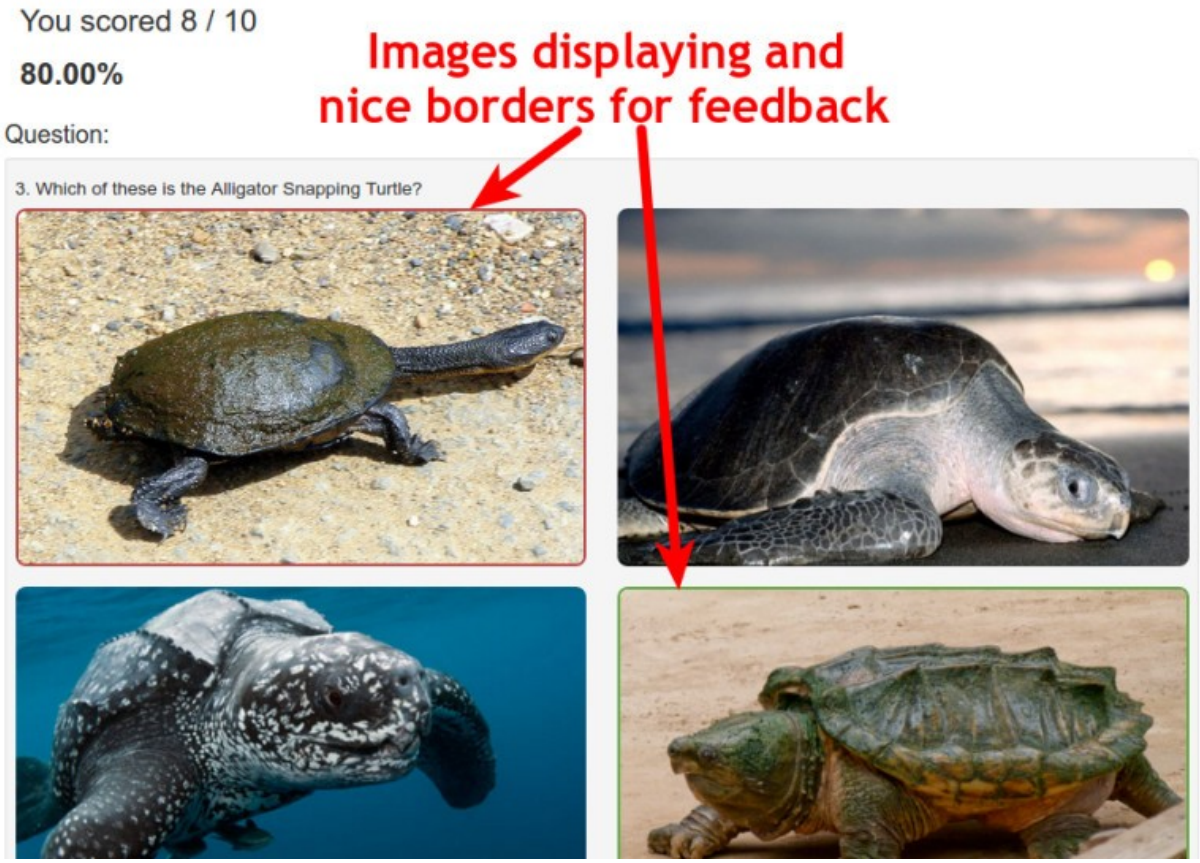
Now on the div with the class image-answer we will also give it an ng-class and pass in the same function we used earlier – getAnswerClass. You may say that this function returns one of two bootstrap classes, neither of which will give the images a border. You would be right.

Doing it this way cuts down on the code logic we need and we can just hook into those bootstrap classes using specificity on top of the image-answer class that the div already has to give it the required borders.

The two classes that the function can choose from is bg-success and bg-danger which are given to correct and incorrect answers respectively. We can now use those and hook into them in our own css.

Add the following to your stylesheet.

```
.image-answer.bg-success{
    border: 3px solid #5ea640;
}
.image-answer.bg-danger{
    border: 3px solid #b74848;
}
```



Return Of The Angular Ng-If Directive

Again, we only want to render one block or the other – either the text answer block or the image block, but never both. This is where ng-if comes into play again. Here are the two blocks with their respective ng-if statements added onto them.

```
<div class="row"
     ng-if="results.dataService.quizQuestions[results.activeQuestion].type === 'tex
t'">

         <div class="col-sm-6" ng-repeat="answer in results.dataService.quizQuestions
[results.activeQuestion].possibilities">
             <h4 class="answer"
                 ng-class="results.getAnswerClass($index)">
                     {{answer.answer}}

                     <p class="pull-right"
                        ng-show="$index !== results.quizMetrics.correctAnswers[result
s.activeQuestion] && $index === results.dataService.quizQuestions[results.activeQuesti
on].selected">Your Answer</p>
                     <p class="pull-right"
                        ng-show="$index === results.quizMetrics.correctAnswers[result
s.activeQuestion]">Correct Answer</p>
              </h4>
         </div>

</div>


<div class="row"
     ng-if="results.dataService.quizQuestions[results.activeQuestion].type === 'imag
e'">

         <div class="col-sm-6" ng-repeat="answer in results.dataService.quizQuestions
[results.activeQuestion].possibilities">

             <div class="image-answer"
                  ng-class="results.getAnswerClass($index)">
                      <img ng-src="{{answer.answer}}">
             </div>
         </div>
</div>
```

You can see that the statements are identical except for the type we are testing for. Image and text.

Now the app is pretty much in its completed state. We just need to give the user a way to return back to the fact page and start everything over again if they wish. Of course, we will need to reset everything when we go back as well, just so that it is a fresh start if the user wants to try again.

Chapter 24

We Have Made It To The End, Friends

There is only one small thing that we still have left to do before you have your finished Angular project. That feature is just adding a button that will take us back to the start – the list

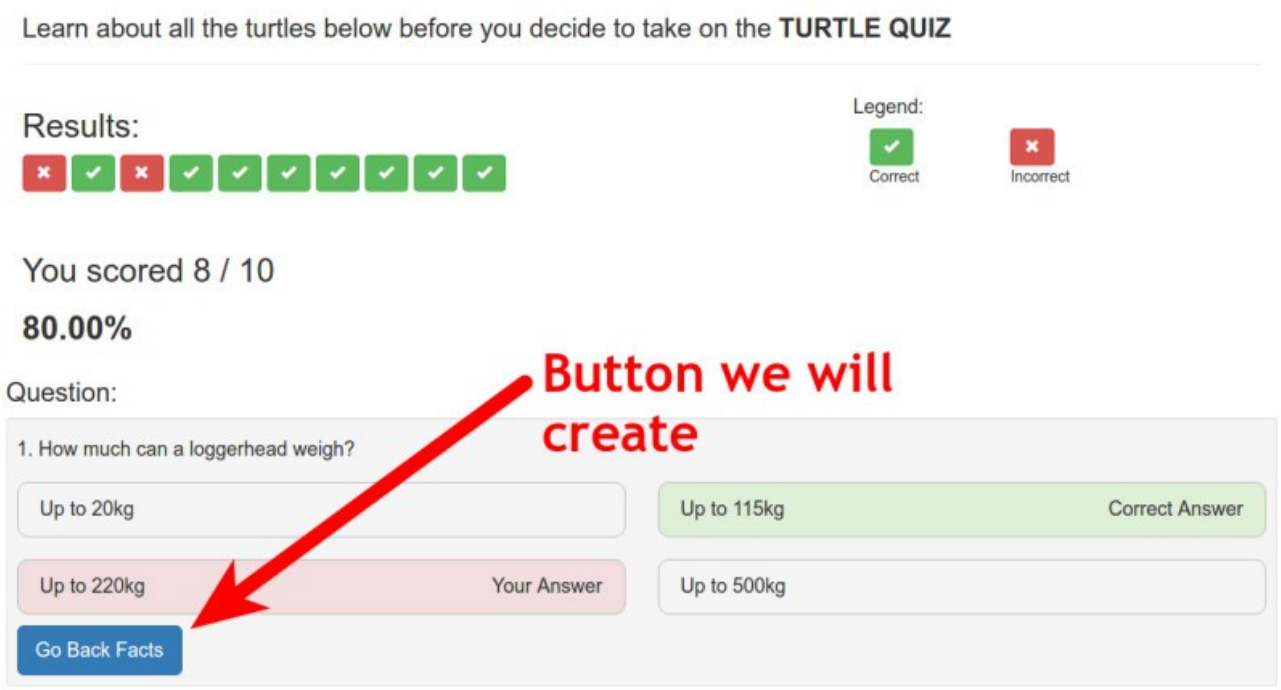
page.

If you want to see the app for yourself, check it out here.

The git repo can be found here.

Click the image to go to the video on youtube or go here https://www.youtube.com/watch?v=EEOMwWO0lhg&index=24&list=PLqr0oBkln1FBmOjK24_B4y_VAA8736wPq

Right underneath the well that holds the questions we will add a large button to take the user back to the facts page. This button will have an ng-click that will call a function that will reset all the data in the application and allow the user to repeat the quiz again without any of the data from their first attempt interfering.




```
<button class="btn btn-primary btn-lg" ng-click="results.reset()">Go Back To Facts</butt
on>
```

Now create the reset function. This function will change the results state back to false and reset the numCorrect property to zero (this is to stop interference on subsequent attempts at the quiz). Then we want to loop through all the questions in the quiz and reset the correct and selected properties back to null.

```
function reset(){

    quizMetrics.changeState("results", false);
    quizMetrics.numCorrect = 0;

    for(var i = 0; i < DataService.quizQuestions.length; i++){
        var data = DataService.quizQuestions[i]; //binding the current question to data

        data.selected = null;
        data.correct = null;
    }
}
```

Yay! A Finished Angular Project!

Well done! You have completed an Angular Project. If this is the first one you have ever made then extra congratulations to you. You have made a huge step; a step that many “developers” never make. They remain armchair developers and don’t actually write code. Which is no way to improve.

I really hope that you have enjoyed this series and that you have learnt a bit about Angular and see just how useful it can be. If you apply some of the knowledge learnt here to one of your own projects I would love to hear about it. Get hold of me via the contact me page on the site or on facebook or twitter.

I look forward to creating many more courses for you in the future and hopefully we can continue to learn from and inspire each other into the future.

If you haven’t already, sign up to my email newsletter where I will keep you up to data with general news in the developer world as well as keep you up to date with all my latest projects, courses and tutorials. Also I would greatly appreciate if you could hit the subscribe button above to subscribe to my youtube channel. I love and appreciate everyone’s ongoing support.

Until my next course, why not check out some of the other content that I have created on the site.

Stay hungry, and keep coding!

Adrian