Angular template

angular has a strong templating engine that can be used to binding controller action and variables, filtering data, binding to DOM events and more. a lot of the power in angular templates comes from the framework core directives. Directives are like HTML extension that provide us add tags or attributes that gives more power to our template. We will cover in this feature Angular templating main features.

Install Angular

we start this tutorial by starting a new angular project.
Create a new directory for your project and in your terminal cd to that directory.
We will install angular by using **bower** so in your terminal type:
> bower init
> bower install angular --save

this will create the **bower_components** folder with the **angular** package.

Create a new **index.html** and place the following **HTML**

```
<!DOCTYPE html>
<html ng-app="AppModule">
<head>
  <meta charset="UTF-8">
  <title>Angular Templates</title>
</head>
<body>
  <div ng-controller="MainController as ctrl">
     <div ng-include="'src/templates/main.template.html'"></div>
  </div>

  <script src="bower_components/angular/angular.js"></script>
  <script src="src/app.module.js"></script>
  <script src="src/controllers/main.controller.js"></script>
</body>
</html>
```

at the bottom of the body we included two scripts.
app.module.js is the script where we create and config our module and for now it will look like this:

```
angular.module('AppModule', []);
```
we will define our controller to look like this:
```
angular.module('AppModule').controller('MainController', function(){
  var self = this;
});
```

we also defined a template for our controller located in **src/templates/main.template.html**
```
<h1>Angular Template</h1>
```

verify that you app is working, and let's explore with this app the powers we have with angular template engine

Interpolation

Using double curly braces in our template we can do calculation:
```
{{ 10 * 5}}
```

we can bind the template to a variable

in the template:
```
{{ctrl.greeting}}
```

in teh controller:
```
self.greeting = 'hello world';
```

call a method in our controller that will return data

in the template
```
{{ctrl.greetingFunc();}}
```

in the controller
```
self.greetingFunc = function greetingFunc(){
  return 'hello from greetingFunc';
}
```

you can also be certain that if your binded variable will change, the template will get updated. For example if I inject **$timeout** (a wrapper around setTimeout) and will change in the **$timeout** the variable **greeting** the template should update as well, let's see that in action:
controller:
```
angular.module('AppModule').controller('MainController', function($timeout){
```

```
  var self = this;

  self.greeting = 'hello world';

  self.greetingFunc = function greetingFunc(){
      return 'hello from greetingFunc';
  }

  $timeout(function () {
      self.greeting = 'hello changed from timeout';
  }, 3000)
});
```

you should see the message is changing in the template.

Two way binding

with angular we can easily bind from inputs to our controller properties and from the controller properties back to the input. Angular refers to this as **two way binding**.
we will try to demonstrate this by creating a form with a username input, this input will be attached to a variable in the controller, our form will also contain a clear button that will call a method on our controller and that method will print the username variable to the input and then will clear it, this will demonstrate the binding from the template to the controller (we are printing the variable before clearing it) and the binding from the controller to the template (we are clearing the variable and seeing this on the template).

place the following code in the bottom of your template:

```
<form name="clearForm" ng-submit="ctrl.clearForm()">
  <label>Username</label>
  <input type="text" name="username" ng-model="ctrl.username" />
  <br/>
  <button type="submit">Clear</button>
</form>
```

We used two directives here, one is **ng-submit** that is used to attach a form submit to controller function, and also prevents the default behaviour of form submission, another directive we attached here is the **ng-model** which binds the input in a two way to a controller property.

In our controller place the following at the bottom of the controller:

```
self.username = 'Enter Username';
self.clearForm = function clearForm(){
```

```
  console.log('The username before deleting: ' + self.username);
  self.username = '';
}
```

Form Validation

with angular we can easily validate our forms by just using the template system. validation in the front end can be disabled so it's not a substitute for backend validation. In angular the value of **ngModel** won't be set if it doesn't pass validation.
Let's work on the form we created earlier and try to check out how we create form validation.

```
<form name="clearForm" ng-submit="ctrl.clearForm()" novalidate>
  <label>Username</label>
  <input type="text" name="username" ng-model="ctrl.username" ng-required />
  <br/>
  <button type="submit">Clear</button>
</form>
```

notice that we placed a **novalidate** in the form tag to prevent the default HTML validation.
we placed the **ng-required** validation in the text input. We can see the following classes are added to the input.

- **ng-valid/ng-invalid -** is the field valid or invalid
- **ng-valid-[key]/ng-invalid-[key] -** whether a certain validation is valid or invalid, in the example above the class that was added: **ng-invalid-required/ng-valid-required**
- **ng-pristine/ng-dirty -** the controller has not been interacted with yet. when you focus and type (even if you kept the value like before) the class will vanish and will be replaced by **ng-dirty**
- **ng-touched/ng-untouched -** notice that at the start of the page the input has a class of **ng-untouched** and when we focus and blur out of the input that class is replaced to **ng-touched**

let's add a mail input to our form. Edit the form HTML to look like this:

```
<form name="clearForm" ng-submit="ctrl.clearForm()" novalidate>
  <label>Username</label>
  <input type="text" name="username" ng-model="ctrl.username" ng-required />
  <br/>
  <label>Email</label>
  <input type="email" name="email" ng-model="ctrl.email" ng-required />
  <br/>
  <button type="submit">Clear</button>
```

```
</form>
```

notice that when you place an input with type **email** , angular will automatically add email validation to that field. So you should see the following class: **ng-invalid-email/ng-valid-email**

FormControl/NgModelControl

when angular sees the form with a name attribute, it creates an instance of **FormControl** to that form, also when there is an **ngModel** directive to an input with a name angular will create an **NgModelControl** and assign it to the controls of the **FormControl** wrapping that **NgModelControl.** with each of those instance that are created you can reference them by using the **name** attribute that you set.
you have the full documentation of **FormControl** in this URL:
https://docs.angularjs.org/api/ng/type/form.FormController
let's try and examine the **FormControl.** replace the form code with the following:

```
<form name="clearForm" ng-submit="ctrl.clearForm()" novalidate>
  <label>Username</label>
  <input type="text" name="username" ng-model="ctrl.username" ng-required />
  <br/>
  <label>Email</label>
  <input type="email" name="email" ng-model="ctrl.email" ng-required />
  <br/>
  <button type="submit">Clear</button>
  <br/>
  <strong>Pristine:<strong> {{clearForm.$pristine}}
  <br/>
  <strong>Dirty:<strong> {{clearForm.$dirty}}
  <br/>
  <strong>Valid:<strong> {{clearForm.$valid}}
  <br/>
  <strong>Invalid:<strong> {{clearForm.$invalid}}
  <br/>
  <strong>Pending:<strong> {{clearForm.$pending}}
  <br/>
  <strong>Submitted:<strong> {{clearForm.$submitted}}
  <br/>
  <strong>Errors:<strong> {{clearForm.$error | json}}
</form>
```

we see here that when we place a name in the form (**clearForm**) angular created a **FormControl** and we can access the attributes of the **FormControl** instance. You can check the **FormControl** class in this URL: https://docs.angularjs.org/api/ng/type/form.FormController

now let's try and explore the **NgModelControl** . we said that by placing a **name** attribute to an input with **ng-model** then the **NgModelControl** is created for us so let's try and modify our form to explore what the **NgModelControl** provides us. Change the form template to look like this:

```
<form name="clearForm" ng-submit="ctrl.clearForm()" novalidate>
  <label>Username</label>
  <input type="text" name="username" ng-model="ctrl.username" ng-required />
  <br/>
  <label>Email</label>
  <input type="email" name="emailControl" ng-model="ctrl.email" ng-required />
  <br/>
  <button type="submit">Clear</button>
  <br/>
  <strong>Pristine:<strong> {{clearForm.emailControl.$pristine}}
  <br/>
  <strong>Dirty:<strong> {{clearForm.emailControl.$dirty}}
  <br/>
  <strong>Valid:<strong> {{clearForm.emailControl.$valid}}
  <br/>
  <strong>Invalid:<strong> {{clearForm.emailControl.$invalid}}
  <br/>
  <strong>Pending:<strong> {{clearForm.emailControl.$pending}}
  <br/>
  <strong>Errors:<strong> {{clearForm.emailControl.$error | json}}
</form>
```

we can access each **NgModelControl** by refrencing the control from the **FormControl** its added to, and you can see it has similar fields that **FormControl** has.
You can read more about the api of **NgModelControl** in this URL:
https://docs.angularjs.org/api/ng/type/ngModel.NgModelController

let's try and add some more validation to our form let's add the **minlength, maxlength,** and **pattern**

```
<form name="clearForm" ng-submit="ctrl.clearForm()" novalidate>
  <label>Username</label>
  <input type="text" name="username" ng-model="ctrl.username" ng-required />
  <br/>
  <label>Email</label>
```

```
  <input type="email" name="emailControl" ng-model="ctrl.email" ng-required />
  <br/>
  <label>
     minlength
  </label>
  <input type="text" ng-minlength="5" name="minlength" ng-model="ctrl.minlength" />
  <br/>
  <strong>Is minlength valid: </strong> {{clearForm.minlength.$valid}}
  <br/>
  <label>
     maxlength
  </label>
  <input type="text" ng-maxlength="5" name="maxlength" ng-model="ctrl.maxlength"  />
  <br/>
  <strong>Is maxlength valid: </strong> {{clearForm.maxlength.$valid}}
  <br/>
  <label>
     pattern - letterns only
  </label>
  <input type="text" ng-pattern="'[a-zA-Z]+'" name="pattern" ng-model="ctrl.pattern"  />
  <br/>
  <strong>Is pattern valid: </strong> {{clearForm.pattern.$valid}}
  <button type="submit">Submit</button>
</form>
```

Angular helps make validation pretty easy for us with the validation directives and with the **FormControl** and **NgModelControl**

let's continue to explore angular template engine with our section on events:

Student Exercise - Forms and Data Binding

- Start a new angular application.
- in the **index.html** add a reference to a new controller with the **ng-controller** and add a reference to a new template with **ng-include**
- in your template add a form to add a TODO task with the same validation that was given to you on the previous task
- use bootstrap to create well designed form with well designed alerts. use the following bootstrap classes:
    - **container, row, col-xs-*, col-offset-xs-*, form-group, form-control, col-form-label, alert, alert-danger, alert-success**
- attach a function in the controller for the submission of the form

- add 2 way binding to the inputs on the form
- in the submission task print all the variables you got from the data binding

Events

angular template helps us easily add event to elements and bind those events to our controller. We already saw one of those events: **ng-submit** but there are a few more to cover.
the syntax to add an event is pretty similar you add an attribute to the html called:
ng-<eventname>=”expression” where expression should map to a control event
let’s try to examine the events directives in angular templates.

ng-click

the **ng-click** directive allows us to add a click event to an element, that element can be a button or other dom element, and the expression of the directive should map to a method in our controllers.
Example of **ng-click:**

```
<h3 ng-click="ctrl.ngClickExample()">
  ng-click
</h3>

<h3 ng-click="ctrl.ngClickExample($event)">
  ng-click with event
</h3>
```

place the code above at the bottom of your template, notice that we can also pass the event object by sending **$event** . you need to add the matching function is your controller so add the following to the controller:

```
self.ngClickExample = function ngClickExample(event){
  alert('ng-click example');
}
```

Mouse events

angular has directives to add mouse events to a dom element. you can also pass the **$event** object to the event you are sending. The available mouse events are:

- **ng-mouseup**
- **ng-mouseenter**
- **ng-mouseleave**

- **ng-mousemove**
- **ng-mouseover**

we will demonstrate **ng-mouseover** which happens when our mouse pointer hover over an element.
Add the following to the bottom of the form.

```
<h3 ng-mouseover="ctrl.mouseOverExample($event)">
  Mouse over
</h3>
```

and in the controller add the following:

```
self.mouseOverExample = function mouseOverExample(event){
  console.log('mouse over!!!');
}
```

try and hover you mouse over the title and see that the console prints the event message.

ng-change

the last event we will cover is the change event which happens when you change an input. add the following code at the bottom of your template.

```
<h3>
  ng-change
</h3>
<input ng-model="ctrl.changeExample" ng-change="ctrl.ngChangeExample();" />
```

and in the controller add the following method:

```
self.ngChangeExample = function ngChangeExample(event){
  console.log('change event');
}
```

there are more events to cover but the ones above are the most common and the rest is pretty much the same syntax as those.

DOM Manipulation

Let's cover some directive that manipulate the DOM tree. Manipulation of the DOM tree means that they can remove certain html elements or add them.

ng-if

The most common DOM manipulation directive is the **ng-if**. the **ng-fi** directive get's an expression that is mapped to a boolean value and will only show the dom element if the boolean expression is **true.** let's check out a simple example. at the bottom of the template file add the following html

```
<h3>
  ng-if
</h3>
<p ng-if="ctrl.isShow">
  This directive will show only if the controller property isShow is equal to true <br/>
  You can toggle the value of isShow by clicking the button bellow
</p>
<button ng-click="ctrl.isShow=!ctrl.isShow">Click me</button>
```

we mapped **ng-if="ctrl.isShow"** and we also added a button that toggles **ctrl.isShow** so when you click the button you should see the paragraph and clicking it again should remove the paragraph from the DOM

ng-repeat

ng-for allows us to iterate on a list of items and create dom changes based on a repeated iteration. A small example of ng-for will be iterating on a list of items from the controller. In your template add the following .

```
<h3>
  ng-for
</h3>
<ul>
  <li ng-repeat="item in ctrl.groceryList">
     {{$index}} {{item}}
  </li>
</ul>
```

```
<form ng-submit="ctrl.groceryList.push(ctrl.groceryAdd);">
  <input name="groceryInput" ng-model="ctrl.groceryAdd" />
  <button type="submit">
     Add Item
  </button>
</form>
```

we are iterating over a list defined in the controller **ctrl.groceryList** and we are printing each list item. Every iteration get's his own scope with a few variables - we are using **$index** to print the current iteration but there are a few more:

- **$first**
- **$middle**
- **$last**
- **$even**
- **$odd**

we also placed a form to add items to our list and we attached in the submit event of the form for the item in the input to be pushed in the array.
In the controller add the array we defined:

```
self.groceryList = ['Milk', 'Cheese', 'Eggs'];
```

reload your app and you should see the list appear, try and items to the list and see how the list automatically grows.
If you needed to do this simple action in **Jquery** what would that require ?

let's try and add a delete button, change your **ng-repeat** to look like this:

```
<li ng-repeat="item in ctrl.groceryList">
  {{$index}} {{item}}
  <button ng-click="ctrl.groceryList.splice($index, 1);">Delete</button>
</li>
```

we added a button with an **ng-click** and in the **ng-click** we are splicing the array based on the **$index**, reload the app and check how the delete button can delete items now and how your list is refreshed.

Pretty cool right ? imagine again about adding the delete button without a framework using just **jquery ?**

Changing appearance

Another popular directive we will use often are directives used for changing the appearance of the app. We will cover here some of those directives.

ng-class

the ng-class determines dynamically what classes should be added to the HTML element. the syntax for the **ng-class** directive is: **ng-class=”expression”** where **expression** can be one of the following:

- a string of class names separated with space
- an object of key as class names and values as boolean expressions, all key with truth will be added
- array of string or objects

let’s add an example of an **ng-class** with string add the following at the bottom of your template:

```
<h2>ng-class</h2>

<h3>ng-class with string</h3>

<div class="ng-class-example" ng-class="ctrl.ngClassExample">
</div>
<form>
  <label>Change ng-class</label>
  <input type="text" ng-model="ctrl.ngClassExample">
</form>
```

we added an **ng-class** attached to the controller variable named **ctrl.ngClassExample**
we also add a form with an input with **ng-model** that let’s you change the **ctrl.ngClassExample** variable.
In the **index.html** let’s add a few styling at the **head** section of our page :

```
<style>
  .ng-class-example{
      height: 100px;
      width: 100px;
      border: 1px solid black;
  }
  .bg-orange{
      background-color: orange;
  }
  .bg-red{
```

```
    background-color: red;
  }
  .bg-black{
    background-color: black;
  }
</style>
```

try and launch your app and put in the input one of the classes above and see how those classes are added to your div.

Let's try and add another example of **ng-class** with an object in the expression.
In your template add the following :

```
<h3>ng-class with object</h3>

<div class="ng-class-example" ng-class="ctrl.ngClassExampleObject">
</div>
<form>
  <label>Change red</label>
  <input type="checkbox" ng-model="ctrl.ngClassExampleObject['bg-red']"
ng-true-value="true" ng-false-value="false">
  <br/>
  <label>Change orange</label>
  <input type="checkbox" ng-model="ctrl.ngClassExampleObject['bg-orange']"
ng-true-value="true" ng-false-value="false">
  <br/>
  <label>Change black</label>
  <input type="checkbox" ng-model="ctrl.ngClassExampleObject['bg-black']"
ng-true-value="true" ng-false-value="false">
</form>
```

in this example we are attaching the **ng-class** to the object: **ctrl.ngClassExampleObject** (which we will soon initiate in the controller) and we attached a form with checkboxes to toggle the object keys, notice that you can control the checkbox values with the directives **ng-true-value/ng-false-value**
in our controller we need to add the **ctrl.ngClassExampleObject** so add this line in your controller:

```
self.ngClassExampleObject = {
  'bg-red': false,
  'bg-orange': false,
  'bg-black': false
}
```

notice that the keys map to class names, and the value map to boolean and if is truthy will place the class in the div.

ng-style

another way other than to control the classes of an element to change it's appearance is to control the style attribute of an element. For this we can use the directive **ng-style.**
the syntax is **ng-style="expression"** where expression is an object where the keys are the styles and the values are the style values.
let's try and demonstrate this directive.

```
<h2>ng-style</h2>
<div ng-style="ctrl.ngStyleObj">
</div>
<form ng-submit="ctrl.ngStyleObj[ctrl.styleKey]=ctrl.styleValue">
  <label>Style key</label>
  <input type="text" name="style-key" ng-model="ctrl.styleKey" />
  <br/>
  <label>Style value</label>
  <input type="text" name="style-value" ng-model="ctrl.styleValue" />
  <button type="submit">Submit</button>
</form>
```

notice that we place a div with the **ng-style** directive attached to the object **ctrl.ngStyleObj**
we also created a form with two text inputs, one controlling the keys of the object and the other the values of the keys. We placed an **ng-submit** directive for the form that adds the key value to **ctrl.ngStyleObj**.
in the controller we need to init the **ctrl.ngStyleObj** with the following line:

```
self.ngStyleObj = {}
```

launch your app and try to put css keys and values and see how the div changes.

Filters

filters format the value of an expression to display the expression for the user in user friendly way. Filters can be used not only in templates but can also be injected in controllers and services. angular comes with a collection of built in filters and we will cover the ones most commonly used. You can also create a custom filter of your own. the general syntax of a filter is:
{{expression | filter:argument1:argument2 }}
you can also chain filter to filter
{{expression | filter1 | filter2}}

dateFilter

formats date to a string based on the requested format.
The format accepts the following:
'y' - year, **'M' -** month, **'d' -** day, **'HH' -** hour, **'SS' -** second
for the full list go to: https://docs.angularjs.org/api/ng/filter/date

let's try and demonstrate this filter, at the bottom of your template file add the following:

```
<h3>
  Date filter
</h3>

{{ ctrl.today | date:'dd/MM/yyyy' }}
```

and in the controller add the **ctrl.today:**
self.today = new Date();
we added a format for the date to print just the date month and year

filterFilter

sounds like a joke but there is actually a filter called filter, and is actually quite useful. The filter selects a subset items from an array and returns it as a new array.

example in the template file place the following code:

```
<h3>
  Filter filter
</h3>

<form>
  <label>Search</label>
  <input type="text" ng-model="ctrl.search" />
</form>
<ul>
  <li ng-repeat="item in ctrl.groceryList | filter:ctrl.search">
     {{item}}
  </li>
</ul>
```

notice that we combined the filter with an **ng-repeat** directive and we are iterating over the filtered array, and basically we implemented a search for our grocery list

jsonFilter

the last filter we will cover is the json filter which is a simple filter that is usefull for debugging which just prints the object in a json string format

Custom Filter

you can create your own custom filter as well, in fact it's recommended to create a filter then to do the filtering in the controller, performance wise. if you filter in the controller the filtering will run every $**digest** loop where in filter the filtering will run only when the input to the filter is changed.
We will now demonstrate creating a custom filter called **zigzag** which takes a string as parameter and will present the string where the first letter will be capital, second lowercase, third capital etc.

in your **src** folder create a folder called **filters** and create a file called: **zigzag.filter.js**. place the following code in the file **zigzag.filter.js**:

```
angular.module('AppModule').filter('zigzag', function(){
  return function(input){
     var result = '';
     for(var i=0; i<input.length; i++){
        if (i % 2 === 0){
           result+=input[i].toUpperCase();
        } else {
           result+=input[i].toLowerCase();
        }
     }
     return result;
  }
});
```

you can see that creating a filter is quite simple. you need to get your module instance and then call the **filter** method to add a filter to our module. The **filter** method accepts the filter name and a factory function that creates our filter, the filter function get's the input as first param and the arguments as the rest and returns the string to display.
You also have to include the new script in the **index.html** file and also in the template call the **zigzag** filter.

{{'hello world' | zigzag }}

Student Exercise - ng-repeat and filters

- goto the server task list url: **https://nztodo.herokuapp.com/api/task/?format=json**
- copy all the tasks and print them in a controller property in the controller you created in the task before.
- using **ng-repeat** print all the tasks on the screen
- use bootstrap to make your app responsive if we are on a medium screen then 3 tasks should be printed in a row, if we are on a small screen then 2 tasks on a row, and if we are on xs screen then one task in a row.
- use bootstrap to print the tasks more beautifully, you can use the following bootstrap classes: **card, card-block, card-title, card-text, btn, btn-primary, container, row, col-xs**
- in each task add a button to delete that task
- add also a search box at the top of the list and create a search box that will filter the below list

Summary

Angular templating engine is quite powerful, we covered here the directives that we most commonly use in an angular directive but there are more directives in the templating engine and a lot to learn. hopefully the guide will be a good starting point.