AngularJS CheatSheet

Hello World

AngularJS enhances HTML by giving it a *dynamic* layer; i.e., raw HTML just displays things, but with AngularJS we can have parts shown according to, say, input boxes and variables (which raw HTML does not have). [See the JavaScriptCheatSheet PDF.]

An example is worth a thousand words! Here's one from the docs:

```
<!doctype html>
<html ng-app> <!-- ⟨0⟩ AngularJS is active for the entire tree -->
  <head>
    <!-- ⟨1⟩ Actually load AngularJS -->
    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js">
    </script>
  </head>
  <body>
    <div>
      <label>Name:</label>
      <!-- ⟨2⟩ The *value* of this text box is named "yourName" -->
      <input type="text"
             ng-model="yourName"
             placeholder="Enter a name here">
      <hr>
      <!-- ⟨3⟩ Actually use the "yourName" value here,
           *whenever* it is updated! -->
      <h1>Hello {{yourName.toUpperCase()}}!</h1>
    </div>
  </body>
</html>
```

This is a *self-contained* **interactive** program: Whenever the input box's value is changed then so is the greeting header "hello . . . !".

Bindings like `{{this}}` are known as *mustache tags, which tell AngularJS that it should evaluate an expression* and insert the result into the DOM in place of the binding. Rather than a one-time insert, a binding will result in efficient continuous updates whenever the result of the expression evaluation changes.

Exercise! Using `<input type="number" ...>`, make a super simple site with two input boxes for numbers and a header that shows their sum whenever a user inputs values. (Solution)

Exercise! Get a twitter handle from the user and show a link to that twitter page.

Warning! "AngularJS" (controllers and `$scope`) has been completely rewritten as "Angular" (components focused) in an effort to simplify it.

AngularJS makes HTML into a language with variables and control flow

AngularJS enhances HTML with variables, filters (explained below), arrays, JSON Objects, and looping & conditional control flow —using JS's usual `&&`, `||`, `==`, etc for Booleans. Later we'll see that functions can be defined in an associated *controller*, then used in the *view*.

```
<html ng-app>
  <head>
    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js">
    </script>
  </head>
  <body>
    <!-- VARIABLES ------------------------------------------------------>
    <div ng-init="firstname = 'Jason'; lastname = 'Jasim'">
      {{ firstname + ' ' + lastname }} <br> <!-- Jason Jasim -->
      <!-- Let's declare a var, but hide the resulting value from view-->
      <p hidden>  {{  middlename = 'J.'  }}  </p>
      <!-- Show the full name: Jason J. Jasim -->
      {{ firstname + ' ' + middlename + ' ' + lastname }} <br>

      <!-- APPLICATION: Keeping count ------------------------------------>
      <button ng-init='count = 0' ng-click='count = count + 1'>
        Increment
      </button>
      <p> Count: {{count}} </p>

      <!-- FILTERS ------------------------------------------------------>
      {{ firstname + ' ' + lastname  |  uppercase }}<br> <!--JASON JASIM-->
      Price : {{ 21 | currency }} <br> <!-- Price : $21.00 -->
      Pi: {{3.14159265358979323846 | number:2 }} <br> <!-- Pi: 3.14 -->

      <!-- ARRAYS ------------------------------------------------------->
      <!-- ["hi","indeed"] -->
      {{ ["hello", "hi", "indeed", "bye"] | filter: 'i' }} <br>
      <!-- Sort alphabetically -->
      {{ ["hello", "hi", "indeed", "bye"] | orderBy: '' }} <br>

      <!-- JSON OBJECTS ------------------------------------------------->
      <!-- Lets make an array of objects ... -->
      {{ people = [ {name: 'kathy', age: 32},
                    {name: 'bob', age: 12},
                    {name: 'jasim', age: 114}
                  ] }}
      <!-- ... and sort according to a field. -->
      {{ people | orderBy: 'age' }}

      <!-- APPLICATION: Search box -------------------------------------->
      <br> Search: <input ng-model="query" />
      <br> Results: {{ people | filter: query }}
```

```
    <!-- LOOPING ------------------------------------------------------>
    <!-- No more hardcoded lists! -->
    <!-- ng-repeat creates a <li> element for each element in the list,
         using the <li> tag as the template. -->
    <ul>
      <li ng-repeat="p in people | orderBy:'age'">
        {{p.name + ' is ' + p.age + ' years old!' | uppercase }}
      </li>
    </ul>

    <!-- CONDITIONALS ------------------------------------------------->
    <!-- Name this inputs box's value 'age' -->
    <input type="number" ng-model="age">
    <!-- "Hello" is shown whenever the above 'age' input is truthy. -->
    <p type="text" ng-if="age"> Hello! <p>
    <!-- This input box shows only if the above, 'age', has value 12. -->
    <input type="text" ng-if="age == 12" placeholder="Your name?">

  </div>
 </body>
</html>
```

`<!-- VARIABLES -->` To run some code when an element is clicked, we use `ng-click`. In our case, we have a variable whose value increases with each user click.

`<!-- LOOPING -->` We are using the HTML tags of `<ul>` (Unordered List) and `<li>` (List Item) to display the list of items in our array.

`<!-- CONDITIONALS -->` When the condition to `ng-if` is falsy, the HTML element is removed from the DOM; otherwise it's added to the DOM.

Note that arbitrary JS expressions will not be evaluated in moustaches; for that, we give the div in a 'controller' (below) which attaches a method to the `$scope`, and in that method we do our arbitrary JS and that method is the one we *can* call in moustaches (within our div). See here for the docs on the permissible expressions.

Moreover, AngularJS provides HTML directives that "already exist" but *know how evaluate {{moustache}} expressions*; e.g., `<a href="#!/go/{{expr}}">Go!</a>` would *not* evaluate `expr`, for that we use `ng-href`.

Bindings can come from intricate JS code! See "MVC" below.

Filters

Filters change a value's presentation; e.g., the `currency` filter displays a number with a "$" symbol and the right number of decimal places, whereas the `filter` filter returns a subset of an array. Other filters include `uppercase` and `lowercase` for formatting strings, and `orderBy` to sort arrays, `json` to show a JS object literally —useful for debugging— and `limitTo` which slices an array; e.g., `arr | limitTo:2:0` gets the subarray of 2 elements starting at index 0; finally, `number:N` rounds a number to N decimal places.

As seen above, filters can be used in both {{expressions}} (using the pipe, '|', operator) and <directives>. We can also define our own filters:

```
<html ng-app="myApp">
  <head>
    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js">
    </script>
  </head>
  <body>
    {{ "hello" | DemoFilter}} <!-- hello woah! -->
    <script> // NEW
      angular.module("myApp", [])
       .filter('DemoFilter', function() {
           return function(input) {
             // The actual custom filter code goes here
             return input + " woah!"
      }})
    </script>
  </body>
</html>
```

Apps

We may have multiple AngularJS apps for the same HTML file, and each app is declared with the syntax `var app = angular.module(appName,dependencies);`. One then furnishes each `app` with its own controllers, filters, directives, etc —e.g., `app.filter(..., ...)` as shown above.

Directives —Essentially "HTML Functions"

The HTML tags starting with `ng-` are AngularJS *directives*. They operate on HTML elements; e.g., from the first section above, the `ng-model` directive binds a name to the value (text) of an HTML element (such an input box); whereas the `ng-repeat` directive repeats an HTML element.

- A page can be associated with multiple AngularJS apps, and `ng-app` indicates where each app starts.
- `ng-init` defines variables and their corresponding values (in the view itself), as shown above, —but this is usually done in a *controller* (later).

We can also define our own custom directives: Below, our custom `ng-speak` directive injects a `<strong>` tag and prepends the text given to `ng-speak`.

```
<html ng-app="myApp">
  <head>
    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js">
    </script>
  </head>
  <body>
    <!-- This becomes:   <strong> Hello, my friends! </strong>  -->
    <div ng-speak> my friends </div>
    <!-- This becomes:  Sorted ... ["a","m","s","u"] -->
    <ng-woah></ng-woah>
```

```
    <script>
      var myApp = angular.module("myApp", [])

      myApp.directive('ngSpeak', function() {
            return {
              // Get the HTML within the directive call
              transclude: true,
              // Replace the div with this HTML template,
              // using the div contents.
              template: `<strong>
                           Hello, <ng-transclude></ng-transclude>!
                         </strong>`
      }})

      myApp.directive('ngWoah', function() {
            return {
              template: `Sorted ... {{letters | orderBy: ''}}`,
              controller: function ($scope) {
                 $scope.letters = "musa".split('') }
      }})
    </script>
  </body>
</html>
```

AngularJS Custom Directives Naming: The *definition* of directives uses `camelCase` (as in `ngSpeak`), whereas the actual *use* of a directive is done with `kebab-case` (as in `ng-speak`). Moreover, custom directive names must start with `ng`.

Notice that there are two ways to use a directive: Both of `<div ng-speak> ... </div>` and `<ng-speak> ... </ng-speak>` tell AngularJS to instantiate a custom directive; i.e., to expand its template which is managed by an instance of the specified controller.

The `template` may `{{evaluate}}` any data defined in the parent scope; e.g., if our div occurs in the scope of a controller. There are several attributes that can be used during a new directive creation; here are a few:

- ◇ `template`: An inline string (HTML) template.
- ◇ `templateUrl`: An (longer) HTML template inside a separate file.
- ◇ `transclude`: Place the child elements of the directive as within the template.
- ◇ `controller`: Furnish the directive with bindings from JS code.
  - We *attached the `letters` binding to the scope* using AngularJS's "dependency injection": We get the scope of the current tag via the special 'service' `$scope`.
  - More on this below, in "MVC".

  Importantly, the `template` contains the presentation logic and uses bindings, whereas the `controller` provides the *context* in which the bindings are evaluated.

Directives allow us to reuse functionality! Similar parts of our HTML code are abstracted out into `template`s and their bindings (i.e., the pieces, values, in which the HTML parts differ) are abstracted out into `controller`s.

- ◇ We just invoke the directive to get the new feature.
- ◇ This means that any alteration to the directive definition, such as the template, propagates to all use sites; no copy-pasting needed.
- ◇ The main view (index.html) becomes more declarative, cleaner.

Components are a 'saner' form of directives that aim to narrow scope.

The MVC Design Pattern —Model, View, Controller

Instead of declaring variables and their values *within* the view, we can do so in a controller *for* the model.

The *MVC* design pattern describes a separation of concerns by splitting an application into 3 parts:

Model	The application data, state: Variables and functions
View	The presentation of the data: HTML, templates, CSS
Controller	The brains/logic that connects the above two.

Tersely put, MVC is the CRUD operations & state, the GUI, and the code to connect the two —e.g., changes in one are propagated to the other.

MVC splits a program into the ("backend, knowledge") data, the ("frontend, visual") presentation, and the logic layers. That is, *the view is a projection of the model through the HTML template, as directed by the controller.*

E.g., in a restaurant, the kitchen with its ingredients and chefs is the model, the menu and dinning table is the view, and the waiter is the controller.

Two-way binding: When we specify which control is associated to which part of a view, AngularJS automatically keeps the two layers in sync. As such, the *controller* consumes user data from the view, uses model utilities to process the data, then displays new data to the user via the view.

The special `$scope` 'service' is used to expose the model to the view —the controller's job is to populate the scope with behaviour. That is, the view can execute any computation, and access any data, that is bound to `$scope` —as such, an app generally has multiple controllers, each for a particular part of the view.

How do we split up an app?

By way of example, the following code...

```
<!doctype html>
<html ng-app>
  <head>
    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js">
    </script>
  </head>
  <body ng-init="prompt = 'What is your name?'">
    <input type=text ng-model="userName" placeholder="{{prompt}}">
    <div ng-if="userName"> "Why hello there, " {{userName}} </div>
  </body>
</html>
```

Can be split up using a *controller*... (We also change from `userName` to an object `user` with a field `name`)

```
<!doctype html>
<html ng-app="myGreetingApp"> <!-- NEW -->
  <head>
    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js">
    </script>
  </head>
  <body ng-controller="PromptController"> <!-- NEW -->
    <input type=text ng-model="user.name" placeholder="{{prompt}}">
    <div ng-if="user.name"> {{greet(user)}} </div>
    <script> // NEW
      angular.module("myGreetingApp", [])
       .controller("PromptController",
         function($scope){
           // One-time setup for prompt
           $scope.prompt = "What is your name?"
           // Whenever the user changes, this function is called.
           $scope.greet = function(user){
             return  "Why hello there, " + user.name
      }})
    </script>
  </body>
</html>
```

(The line `<body ng-controller="PromptController">` lets everything in the `body` tag have access to the functionality of the `PromptController` (which is just a JS function). Notice that our controller *defines* `prompt` and `greet` for use in the view —*using* the value of the input box, which is declared in the view with `ng-model="user.name"`; conversely, we can update the `user.name` *input box*'s value using the controller: `$scope.user.name = "hola"`.)

The above is a bit messy, and so we split it up further into two files: The `mvc_frontend.html` file focus on presentation only...

```
<!doctype html>
<html ng-app="myGreetingApp">
  <head>
    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js">
    </script>
    <script src="mvc_backend.js"></script> <!-- NEW -->
  </head>
  <body ng-controller="PromptController">
    <input type=text ng-model="user.name" placeholder="{{prompt}}">
    <div ng-if="user.name"> {{greet(user)}} </div>
  </body>
</html>
```

...and the `mvc_backend.js` file, which has the bindings,...

```
// Model
var app = angular.module("myGreetingApp", [])
function greet(user) { return  "Why hello there, " + user.name }

// Controllers
app.controller("PromptController",
      function($scope){
        $scope.prompt = "What is your name?"
        $scope.greet  = greet
     })
```

It may be useful to know that within a controller, `$eval` acts like `{{...}}`; e.g., instead of `$scope.a + $scope.b` we may write `$scope.$eval('a + b')`.

Number Guessing Game

Exercise: Using `ng-init`, `ng-click`, controllers, and other directives, build a "number guessing game"; the user makes a guess in an input box, sees whether the guess is too high or low or correct, and has a "new game" button. Finally, give the user an "I give up" button which shows the answer —hint, use `ng-if`.

Hint: This can be done in under 40 lines, in a single file.

Single Page Applications (SPA) and `$route`

SPA's give the feeling that the entire application is in a single webpage: When the user requests something, *part of the webpage* changes rather than loading an entirely new webpage. Example SPA's include Gmail and Netflix.

That is, instead of having a webpage for each aspect of your app, we have a single webpage and *multiple* views —one of which is shown to the user depending on what they want— each with its own controller (since each view has its own goals).

SPA's can be created as follows —and exemplified in the code below.

1. Add `angular-route` as a script reference *after* the AngularJS script.

   Routing is used to present different views to the user on the same web page: The `$route` service watches the URL, and whenever it changes, a route definition is looked up (i.e., view + controller).

   // [1b] In the app declaration (the JS file), we need to add a dependency on the `ngRoute` module.
2. Add `href` tags which will represent links to the different parts of the application.
3. Add the `ng-view` directive within the application; it's used to inject the various views.

   Whenever the route (the URL in your browser's address bar) changes, we load the associated HTML template view at the location of the `ng-view` directive.
4. Define routes through `$routeProvider`.

   // [4a] Whenever the user clicks on the href tag `home`, from step 2, we place the HTML `About <em>Us</em>` into the place of the `ng-view` tag.

// [4b] Whenever the user clicks on the href tag `"hello"`, from step 2, we place the contents of `hello.html` into the place of the `ng-view` tag.

// [4c] Whenever the user clicks on the href tag `newevent`, the `newevent.html` template is inserted in-place of the `ng-view` tag *and* this view is given the `NewEventController` controller to manage its state and logic.

// [4d] This route definition has a custom key-value pair defined in the route, which is then accessed in step 5 from the `$route` service and attached to the `$scope` for use in the template.

// [4e] Parameters can be passed to the route via the URL; e.g., `index.html#!/go/1` and `index.html#!/go/23` are two routes with the parameter being 1 and 23. The controller service `$routeParams` then access the parameters. (Note: `index.html#!/go/` is the route `go` with an empty parameter; it is different from `index.html#!/go`, which has no parameters and so is either supported in its own route definition or redirects to the default route.)

// [4f] Whenever the URL has a route not accounted for —such as when the page opens up, or the user changes the URL to gibberish—, then we redirect them to a default route. When not provided, unconsidered routes redirect to the landing page; i.e., `redirectTo: '/'`.

5. Add controllers to the application.

   Just as the controller `$scope` service passes information from the model to the view , the `$route` service accesses properties of the route.

   Note that we could reuse the controller among multiple views, or, more likely, each view could have its own controller.

6. Create the different web pages for the application

   In our example below, we use the following templates:

hello.html
```
Hello!
```

newevent.html
```
Wanna add a new event, eh?
{{message}}
```

. . . . . index.html . . . . .

```
<html ng-app = "myApp">
  <head>
    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js">
    </script>
    <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular-route.min.js">
    </script>  <!-- [1] -->
    <script src="script.js"></script>
  </head>
  <body>
    This text appears alongside every view <hr>
    <!-- [2] -->
    <a href="#!/hello">Hello</a><br>
    <a href="#!/home">Home</a><br>
    <a href="#!/newevent">Add new event</a><br>
    <a href="#!/greet">Welcome</a><br>
    <input type="number" ng-model="it" placeholder="0">
    <a ng-href="#!/go/{{it}}">Go!</a>
    <!-- [3] -->
    <br><div ng-view></div><hr>
    Likewise, this text appears alongside every view
  </body>
</html>
```

. . . . . script.js . . . . .

```
var myApp = angular
            .module("myApp",["ngRoute"]) // [1b]

myApp.config(function($routeProvider){ // [4]
    $routeProvider
        // [4a] Inline HTML template
        .when("/home",{
            template:"About <em>Us</em>"
        })
        // [4b] HTML template in another file
        .when("/hello", {
            templateUrl: "hello.html"
        })
        // [4c] HTML template that has {{expressions}}
        .when("/newevent", {
            templateUrl: "newevent.html",
            controller: 'NewEventController'
        })
        // [4d] HTML /inline/ template that has {{expressions}}
        // and has a key-value pair declared in the route definition.
        .when("/greet", {
            myfriend: "Jasim",
            template: "Hello, {{person}}!",
            controller: 'GreetingController'
        })
         // [4e] A template with a parameter "myParam";
         // which the controller decides to bind it to "page".
        .when("/go/:myParam", {
            template: "Welcome to page: {{page}}",
            controller: 'GoController'
        })
        // [4f] When gibberish is appended to the URL, go to "/hello"
        .otherwise({
            redirectTo: '/hello'
        })})
// [5]
myApp.controller('NewEventController', function($scope){  // [4c]
    $scope.message = "Hola!" })
myApp.controller('GreetingController', function($scope, $route){ // [4d]
    $scope.person = $route.current.myfriend })
myApp.controller('GoController', function($scope, $routeParams){ // [4e]
    $scope.page = $routeParams.myParam })
```

Things may not work! Open up the console in your browser —F12— and you may see an XMLHttpRequest (XHR) error; XHR requests is how AngularJS loads templates via an HTTP server. One solution is to run your own web server:

1. `npm install http-server -g`
2. `cd ⟨my-folder⟩`
3. `http-server`
4. Then open `http://localhost:8080/index.html`

Reads

The first two links below each contain lots of small, digestible, tutorials on AngularJS with numerous screenshots.

- guru99.com ;; AngularJS Tutorial for Beginners: Learn AngularJS Step by Step
- wikitechy.com ;; My First Application In AngularJS
- w3schools ;; AngularJS Tutorial
- Youtube ;; Introduction to Angular.js in 50 Examples
- What is MVC, and how is it like a sandwich shop?