Angular.JS Created 2009 by Google https://angularjs.org/

Angular style guide:

https://github.com/johnpapa/angular-styleguide/blob/master/a1/README.md
https://github.com/johnpapa/angular-styleguide/blob/master/a2/README.md

Angular 1:
Rule of 1
Define 1 component per file, recommended to be less than 400 lines of code.
Why?: One component per file promotes easier unit testing and mocking.
Why?: One component per file makes it far easier to read, maintain, and avoid collisions with teams in source control.
Why?: One component per file avoids hidden bugs that often arise when combining components in a file where they may share variables, create unwanted closures, or unwanted coupling with dependencies.

Small Functions:
Define small functions, no more than 75 LOC (less is better). (LOC = Line Of Code)
Why?: Small functions are easier to test, especially when they do one thing and serve one purpose.
Why?: Small functions promote reuse.
Why?: Small functions are easier to read.
Why?: Small functions are easier to maintain.
Why?: Small functions help avoid hidden bugs that come with large functions that share variables with external scope, create unwanted closures, or unwanted coupling with dependencies.
More on the link above for angular 1

Angular 2:
If you are looking for an opinionated style guide for syntax, conventions, and structuring Angular applications, then step right in. These styles are based on development experience with Angular, presentations, Pluralsight training courses and working in teams.
https://angular.io/guide/styleguide

Introduction and Bascis of AngularJS

Extends HTML attributes using Angular 'directives'
Two-way data binding is at the core of Angular. When data is changed / entered on a form this goes through to the model, and vice-versa.
It's a **client side JavaScript framework** (we don't need to know what system the users have, what device, it should run fine. because it's JavaScript, it runs entirely inside of a web browser. We're limited by what the web browser allows us to accomplish.
The code is executed in the web-browser, so it's really fast and responsive.
We don't need to save data, or serve a page, other elements will do that, not Angular. **We just render the view on the screen. We focus on the V of MVC.**
We need to assume the user will be using the buttons and functions of their browser (we need to program this functions so it works for them as expected. This allow the user to do as much as possible on their device (whatever it might be) before sending it back to the server.

Controller = javascript functions

Model - JS objects/properties

View: Document Object Model

Angular extends the DOM (by using directive). We just define further objects in that model.

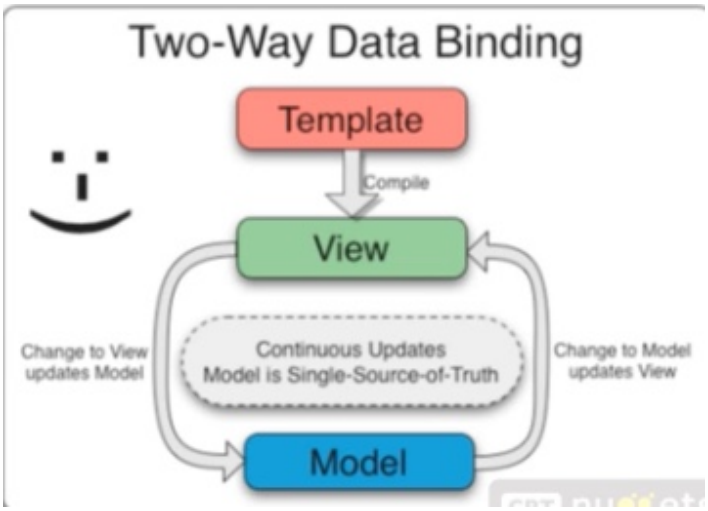
```
<input ng-model="person.name" type = "text" placeholder="Your Name" />
<h1> Hello {{ person.name }} </h1>
```

ng-model is a directive says: save that value for later. I don't have to declare that variable.
The ng-model directive, or HTML attribute, allowed us to tie an input box back to a variable in our background, so that whatever we typed into that input box was stored right inside of that variable, and we had access to it.

Data binding in Angular

Data binding is getting the information, getting the data, out of the program, out of memory, on a computer and on to the screen so the user can see it and manipulate it if they need to.

For it to work, just add a script tag in the html code with a link to angular library.
and at the top as well **<html ng-app="myApp">**



The model is the only source of truth. The view reflects it. They stay in syncs.

ng-repeat: allow to add a for each loop in html

```
<div ng-controller="SampleCtrl">
	<table>
		<thead><th>Last</th><th>First</th></thead>
		<tbody>
			<tr ng-repeat='person in people">
				<td> {{ person.last }} </td>
				<td> {{ person.first }}</td>
			</tr>
			<tr>
			<td><input ng-model="addLast" type="text" placeholder="Last" /> </td>
			<td><input ng-model="addFirst" type="text" placeholder="First" /> </td>
			<td><button ng-click="addPerson()">Add person</button> </td>
			</tr>
		</tbody>
	</table>
</div>
```

In another file (to provide dummy data to populate the table / it should come from a database):

```
function SampleCtrl ($scope) {   // $scope is a dependency injection
	$scope.people = [
		{last: "Krus", first: "Martin"},
		{last: "Krus", first: "Joanne"},
		{last: "Krus", first: "Zoé"}
	];}
```

While the user is experiencing something very responsive on screen, in the background the changes to the database might be slower, but the user doesn't know that.

Controllers

in the html, in a div, name the controller: ** ** This is the only place, within that div, where the action of the controller is defined)
and in a js file, define the controller:

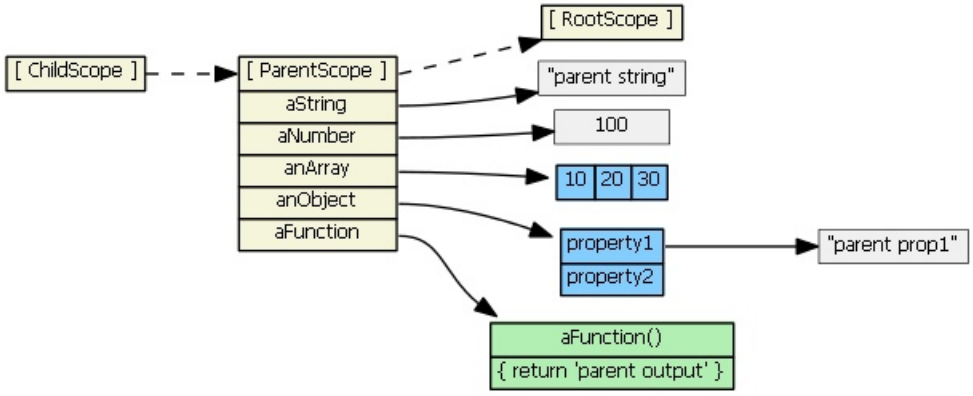
```
function SampleCtrl ($scope){
	$scope.changeName = function() {
		$scope.person.name = "Ben";
	};};
```

To do this better with module:
var myApp = angular.module('myApp',[ ]);
myApp.controller('HelloCtrl', function($scope) {
define the rest }

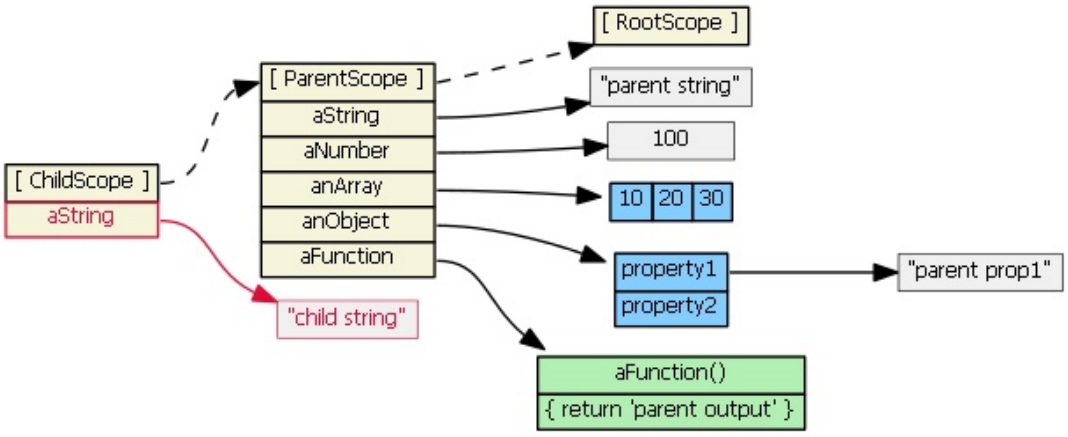
AngularJS uses controllers on the client side versus the server side to enable faster, more responsive, more dynamic web applications.

Managing Scope with AngularJS

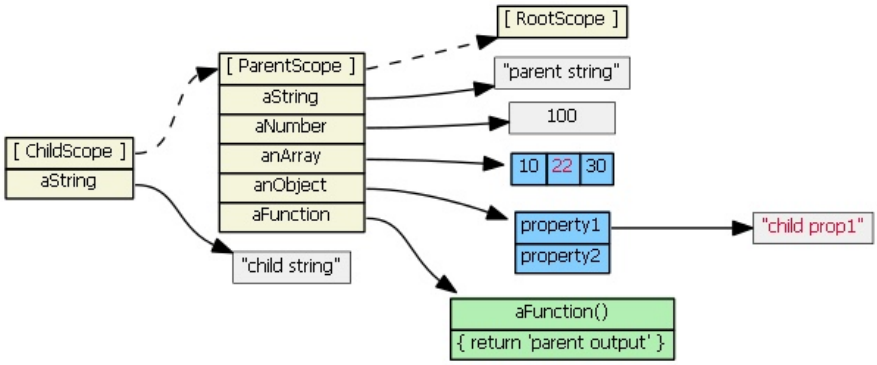
prototypical inheritance: https://github.com/angular/angular.js/wiki/understanding-scopes



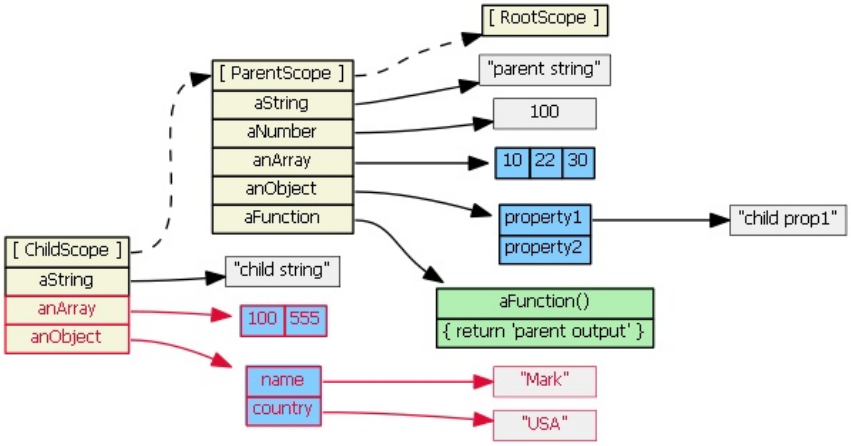
childScope.aString = "child String"
The prototype chain is not consulted, and a new aString property is added to the childScope. This new property hides/shadows the parentScope property with the same name.



childScope.anArray[1] = 22 it goes back up and change the value 20 to 22 even for the parent.
childScope.anObject.property1 = 'child prop1' (there is two dots)
The prototype chain is consulted because the objects (anArray and anObject) are not found in the childScope and we are accessing a part of them. The objects are found in the parentScope, and the property values are updated on the original objects. No new properties are added to the childScope; no new objects are created. (Note that in JavaScript arrays and functions are also objects.)

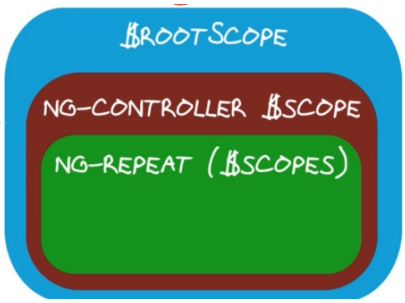


childScope.anArray = [100, 555] (this time not accessing an element of the array but giving the array a new value)
childScope.anObject = { name: 'Mark', country: 'USA' } (not accessing a prop of the object but changing the object) - only one dot.
The prototype chain is not consulted, and child scope gets two new object properties that hide/shadow the parentScope object properties with the same names.



Angular provides a really robust service called $SCOPE that allows you to create an application architecture that doesn't natively exist in JavaScript. Angular dynamically builds it for you. And it does it in a couple different ways. The primary way that you run into is when you do an NG-CONTROLLER directive. NG-CONTROLLER very simply creates a new scope object whenever you define an NG-CONTROLLER inside your HTML template.
The **scope service** has a couple of different responsibilities inside of Angular. Underneath the hood, it's being watched by something called the **watch service**. And what that's doing is looking for any changes on the scope. And when it sees them, **it's calling the digest loop in order to update the model or the view as necessary based on any model mutations, any changes to the values of your model**. Because of this, you always want to assign your values to the scope object itself inside of your controller.
You also want the scope to not only be a part of this digest loop, you want it to be the object that contains your model.

Scope objects in Angular do not exist in a bubble. They are actually part of an inheritance hierarchy. **Scopes can actually have children scopes in them**. And a lot of different directors will create scopes inside of scopes. You can create controllers inside of controllers, or other things.
Like NG-REPEAT will actually create multiple scopes inside of that single controller scope. They are children scopes of the parent scope from NG-CONTROLLER. And this means that they inherit the values. They inherit those properties prototypically from their parents.
So when we use NG-REPEAT, we are inheriting from the NG-CONTROLLER scope that was created for NG-CONTROLLER.



The rules still apply, though. So w**e want to make sure that when we're referencing values, either on those children or on those parents**, we're doing it explicitly through those objects with **those additional dots**, person.name, so that we get the exact value that we want from the exact scope object that we want.

Now, all of these scopes, regardless of how many children or children of children scopes you have or don't have inside of your application, all r**oll up to a single parent scope that's in charge. And it's called the root scope.** The root scope is a specific instance of the scope service that always exists inside of an Angular application. It's created by the NG-APP directive. Recall that to load an Angular application, you've got to call this NG-APP directive at some point in your HTML template. This is what bootstraps the Angular application into memory, and it creates the rootscope scope service.

Two events that propagate events up and down your scope hierarchy.
$emit up to the parent
$broadcast down to the children

Expressions {{ }} and | filters:

Expressions are Angular statements inside of your HTML templates that get interpreted by AngularJS. They're embedded right inside of your HTML document, but they don't print out on screen like that. They get executed by Angular and interpreted, and the result is put out on screen. It's how you embed data binding right into your HTML document. Expressions also have the ability to have filters applied to them. Those are functions that you can apply to that expression after Angular is done interpreting it.

Angular, act on what is inside the {} if arithmetic, it will make the calculation before display
no controls, no conditional allowed inside
it's forgiving: undefined object won't cause error but will transform it to null
have access to filters (functions applied directly to the expression |) filter can have arguments.:
 1+2 = {{1+2 | currency: "USD$"}}

more on filters here: https://docs.angularjs.org/api/ng/filter
Will help format, dates, numbers. currency etc...order by, JSON format.
You can even create your own filters.

Extending forms:

Receive data from the user, transform and save in our database.
Form model binding:
in the html various input with ng-model attribute with user.name user.email etc... ng-click="update(user)"
in the controller: $scope.update = function(user) {
$scope.master = angular.copy(user);
};
ng-dirty: data of a form not submit yet, **valid** when the type of what is entered match the type of the input (email, text etc..) declared, otherwise **invalid**
In the CSS:
input.ng-invalid.ng-dirty { background-color: red; }
input.ng-valid.ng-dirty { background-color: green; }

To go a bit further: http://jsfiddle.net/uXnBP/
We reference $scope.myform because by giving the form the name attribute, I've created a myform object inside of my scope that I can access. Same thing with the username on the input box. This has created a username object inside of my form, in my code, that I can reference inside of the controller.

This different than the user.name that's hooked up to the ng-model, as user.name is only storing the value of the input box. Whatever's keyed into the input box is being stored into user.name. It doesn't have anything else, any other functionality associated with it.It's just the model.
This **username object**, on the other hand, **has access to a whole bunch of other codes, such as the error service** and the **required** value. So here I can test on whether or not this error is true, if it is invalid because of the required variable. And if it is, I can alert, please enter a username.
Now, because those objects-- myform and username-- are defined inside of the $scope container, the model container for our controller, we can actually reference those objects up in our HTML template using Angular directives and expressions to create error message for example only showed if invalid:
<div ng-show="MyForm.userEmail.$dirty && !form.userEmail.$valid" etc... with span

That way the controller only stores the state of the object and the view is dealing with the errors itself.

form validation

with the number type, name=num , if using min and max, we can use ng-show="myForm.num.$invalid
$error.url works with invalid url format
We can use custom validation too (see custom directives later)

Directives

Directives are the control statements inside of your DOM, inside of your HTML code, that tell AngularJS how that DOM element, how that HTML, should look or behave.
They are more than just custom attributes or angular attributes. AngularJS as other ways to reference a directive.
ng-model max min are directives, it's like attribute, and there are different way to create directives:

<input ng-model="xyz' ... <input type='text' MAX='5' ... <input class='new-directive'.....
<new-directive> </new-directive> recommended (the two in bold) they make more sense than the others.

vocabulary:
DOM = page
Element = <....> < />
Parent / Child: is the relationship between the various elements
Attribute: are the code inside those angular elements, they belong to that element
Value: this was exists in between the element statements, this is the data.

denormalized / normalized
html is not case sensitive - javascript is
in html ng_repeat ng-repeat ng:repeat...will all be normalized in angular as ngRepeat to be processed

Why directives? because they are : concise, extensible, testable (they create a good separation of concern MVC)

example of build in directives: https://docs.angularjs.org/api/ng/directive
ng-app ng-controller ng-model ng-bind ng-click ng-repeat ng-switch ng-blur
input text date email number form select textarea and more (html elements that have been over written)

For custom directives:
factory: moduleName.directive(name, Factory)
Two way to declare a custom directive: **linking function** or **directive definition object**
linking function only:
return a function (
scope --> $scope
element --> <..> </...>
attributes [:] pair of key value related to the element
controller)
example:

```
myApp.directive('myForm', function () {
        var linkFn = function(scope, elem) {
                scope.count = 15;
                elem.on ('click', function(){
                        scope.$apply(function() {
                                scope.count++;
                        });
                });
         };
        return linkFn;
});
```

et dans le html:

```
<div> my-form>
        <p> Click count: {{count }} </p>
</div>
```

https://docs.angularjs.org/api/ng/service/$compile
For more powerful directive: **directive definition object** **DDO** (best practice) list of parameters:
scope: true/false (if we want to create a scope or not
controller : function ()
restrict: EACM
String of subset of `EACM` which restricts the directive to a specific directive declaration style. If omitted, the defaults (elements and attributes) are used.

- `E` - Element name (default): `<my-directive></my-directive>`
- `A` - Attribute (default): `<div my-directive="exp"></div>`
- `C` - Class: `<div class="my-directive: exp;"></div>`
- `M` - Comment: `<!-- directive: my-directive exp -->`

Template: string of html elements
templateURL: string of the url to a file of html element which is the template
compile: function()
link: function()

example:

```
myApp.directive('myForm', function () {
        var linkFn = function(scope, elem) {
                scope.count = 15;
                elem.on ('click', function(){
                        scope.$apply(function() {
                                scope.count++;
                        });
                });
         };
        var ddo =
        {
        scope: false, (if true, create a new child scope seperate from the parent)
        link: linkFn
        };
        return ddo;
});
```

et dans le html:

```
<div> my-form>
        <p> Click count: {{count }} </p>
</div>
```

More in this link: https://docs.angularjs.org/api/ng/service/$compile

Dependency injection

This is a design pattern (like factory, singleton etc...)
list of services: https://docs.angularjs.org/api/ng/service

AngularJS Routing

single page routing
In a multi page, the application state (along with other things) is usually passed by the post request on client side to the server side.
In a single page, only small piece of that page are posted and a little is sent back (less internet traffic and the application state is store on the client side, no need to pass it to the server).
Angular simulates the routing behaviour on the client side like if they were on a server.

$routeprovider, this is part of the ngRoute module. Along with the .when function we are going to use.
.when has a template, template/url attribute, resolve, redirectto, reloadonSearch, controller

Example:

```
var myApp = angular.module('myApp', ['ngRoute']);
myApp.config(function($routeProvider){
        $routeProvider.
                when('/', {template: 'This is the base uri template'})
                .when ('/aRoute', {template: 'This is a route!'})
                .when('/aDifferentRoute', {template: 'This is a different route!'})
                                // passing an object with the just a template which is a simple street
});
```

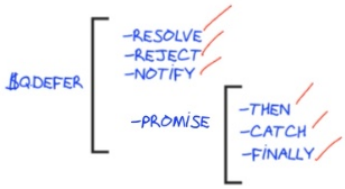
in the html:

```
<div ng-app="myApp">
        <div ng-view></div> // this is where the code is rendered (just the string in that context)
</div>
```

The scope in the controller does not persist across views. Routing actually deletes and recreates the scope every time it loads a view. That means that we cannot use the scope inside of our controller to persist data from view to view. We have to use **a service.**

Promises in Angular

Promises are another design pattern, they focused on dealing with asynchronous calls to remote servers, specifically asynchronous calls that we don't want to wait for finish, but we do want to get notified when they complete.
http://wiki.commonjs.org/wiki/promises
and specifically this one: http://wiki.commonjs.org/wiki/Promises/B
To make the code easier to read.
'A promise is an object that represents a task that may or may not be complete'
The API of angular uses $Q for promises.
you create a $Q.defer object which has: resolve (tell the promise this resolved successfully) - reject (promise should be reject with an error) - notify (triggers the promise to run the notify callback) - promise (is also an object, has itself -then -catch - finally)



you can daisy chain a promise on another promise:

```
promiseB=promiseA.then(
        successCallback: function(){},
        errorCallback: function() {},
        notifyCallback: function() {

);
```

Unit Testing

Unit testing is all about breaking out your code or your application into individual components for testing.
AngularJS is an API that strongly supports the separation of behavior, the separation of code.
unit testing script: assertion (what we want to accomplish) expectation (the result we expect to occur with the assertion and variables we provided)
Jasmine is a JavaScript library that allows us to write assertion-driven unit testing scripts, programmatic-testing scripts against our JavaScript code in JavaScript itself.

example:

```
describe('myFunction', function() {
        it('CalculateAge', function () {
        expect (myFunction('1/1/1990')).toEqual(24);
        expect (myFunction('1/1/2030')).toEqual('Error: Out of range');
        });
});
```

Jasmine is going to work really well with AngularJS. But we are going to run into problems when trying to unit test AngularJS. Namely, we need to bootstrap the AngularJS library. We need to instantiate things like the scope on a controller or the dependency injections for a directive.
Because unit testing happens individually on units outside the context of our application, we're not going to have Angular available to do all those things for us. We need to do it ourselves. And Angular provides a library, called **ngMock,** that's going to help us do that.