**Angular 1.6 Notes (MitAgator Frontend)**

**MVC**:

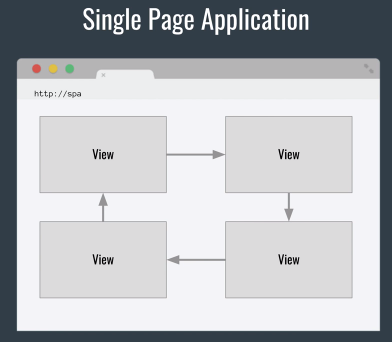
* View is the UI & what it displays. Model is the data (upon which the view is reliant). Controller mediates between the two. So model is your data store, view just displays it.

**VVM (Model View View Model):**

* The view model mediates between the view and model and keeps the in sync with 2-way binding. If one changes the other does also.
* Built into Angular.
* POJO – plain old javascipt object.
* The controller manages the view model which updates the view and model.

**Single page application**:

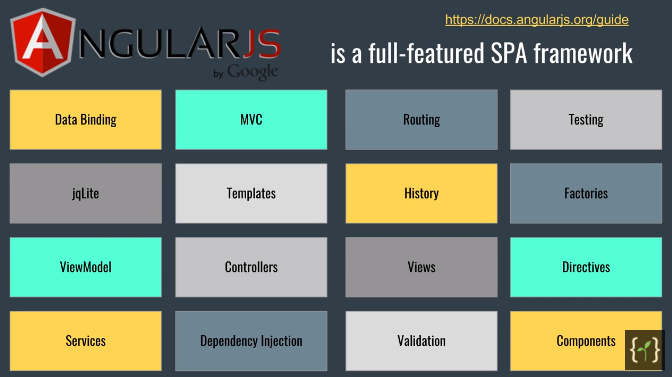
* In Angular you don’t refresh pages – you’re always on the landing page (with the same header and footer). As you change things the middle-section is dynamically swapped in and out to represent the view.
* Your base URL ([www.gmail.com](http://www.gmail.com)) will stay the same, but fragments on your url will change (gmail.com/r/ccc or something).
* **Routes**: A mapping between a view and a controller. When the URL changes to X the router will say please load in this specific view and this specific controller.
* Gmail is an angular application.
* Benefits: You don’t need to send repeated requests to the server, and you don’t need to reload thins that are common to both pages. You’re sending less data from the server, and there’s generally less thinking – the server does no rendering, it just sends JSON back to the frontend where Angular renders the site.
* Angular uses extremely RESTFUL APIs – get data, post data, delete data.
* So, if I click a button only a very small number of things actually change. 90% of the site is identical, but we might repopulate a list of emails because we’ve swapped from ‘inbox’ to ‘sent’.



**Challenges of SPAs:**

* DOM Manipulation – you’ll need to manipulate the DOM to remove/insert/reload your elements as data changes.
* Object Modelling – you’re getting data in from an API and you need to model it in some way.
* History – Hijack the back button on the browser, because we’re always on the same page but we need to simulate a history of page state changes!
* So forth… lots of stuff Angular gives you 😊

**Angular**:



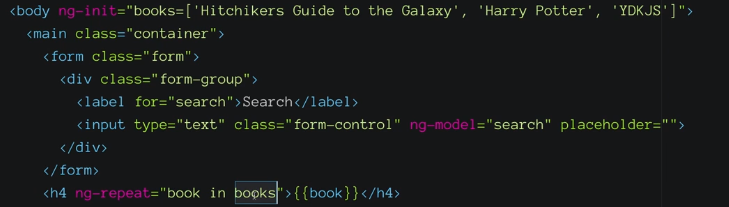
**Directives & Filters:**

**Directives:** Directives are Angular markers on HTML objects. That tell the AngularJS HTML compiler to attach a specific behaviour to that DOM element. Note that you can create custom directives. <https://docs.angularjs.org/guide/directive#what-are-directives->

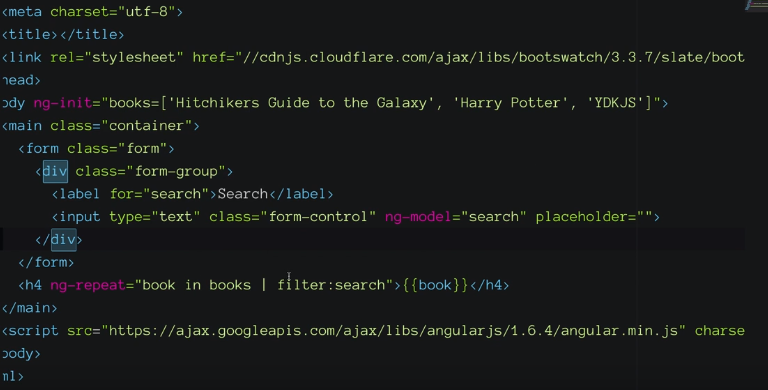
* **ng-model**: Set a placeholder variable directly against an HTML element. There’s no controller here – it’s just basic data binding.



* **ng-init**: Create a variable, i.e. ‘<body ng-init=”books=[‘A’, ‘B’, ‘C’]>
* **ng-repeat**: Takes an iterable/collection (i.e. array) and repeats it. Here we’re saying ‘create an h4 for everything in the declared collection in our model ‘books’.



**Filters:** Filters format the value of an expression for display to the user. Here we’re using the built-in one, saying ‘filter on the search value that is in the scope on the model’. As search updates, our ng-repeat updates to only display the things that match the filter.



**Views, Controllers, and Scope:**



**$scope**

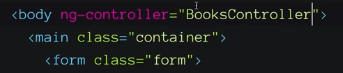
A variable that defines the view model. When you declare ‘ng-init=books’ it’s actually tying the books var to $scope. $scope isn’t something you often directly work with, but it’s basically our view model. **It’s the glue between a controller and a view**.

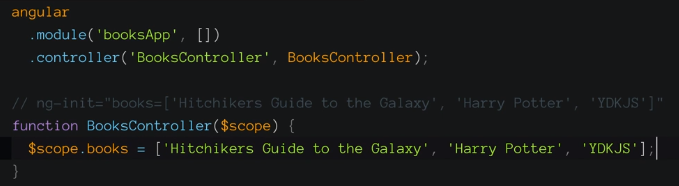
**Modules:**

Defines where controllers, services, filters, etc sit. Initially you’ll have one module, but you can split it out to keep things separately. Different site functionality is wrapped up into a module, any of which might have lots of controllers.

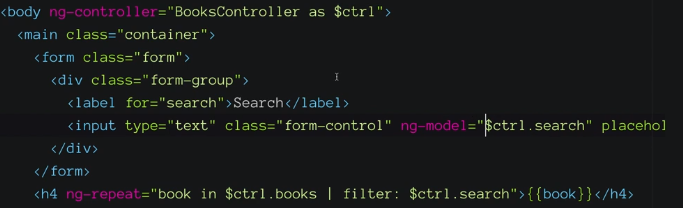
**Initialising a Module:**

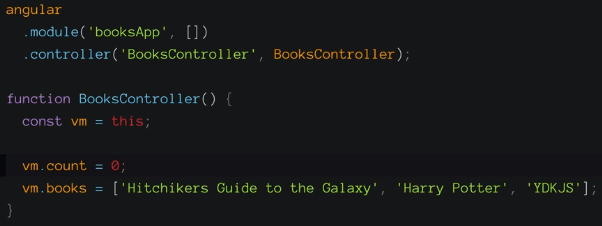
1. **Angular.module()** takes a name and a list of dependencies. Use it in a new file to initialise a module.
2. Inside **ng-app** we need to say ‘anything that is a descendent of this element can use the features/modules/controls in booksApp, our module.
3. We now create a controller within that same file. This controller needs to be tied back to our HTML page – now anything in the body has access to whatever’s in the BooksController.



1. We want to create vars in the JS that our angular frontend nodes can see. We’ll define our new controller and pass it $scope – this will let us assign vars to the $scope object (remembering that’s our view controller).
2. **Updating our code**: – We can also give names to controllers to make things clearer. We also create a new constant with ‘this’ and then add it to the controller’s scope. It’s still going on $scope, but we’re hiding it behind a nicer interface. So here ‘vm’ in the javascript corresponds to the $ctrl model in the HTML.

* By having everything in our contrlller behind a variable we skip a whole bunch of potential bugs – it’s not required, but it’s one style of good form.
* Note – once you’re using ‘controller as x’ you need to be declaring things in the JS against ‘this’. We’re using vm = this for clarity.

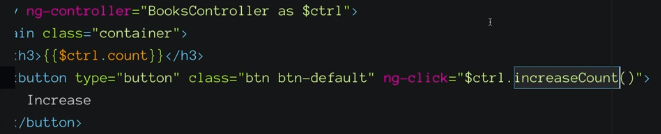




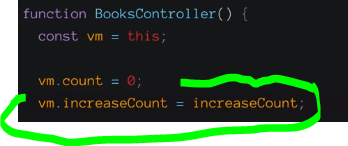
**Defining functions in a controller:**

We’re going to define a function against this button, because we want the count to increment every time it’s clicked.

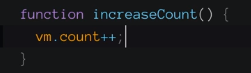
We’ll add our ng-click directive to the DOM element and set it to fire a function we’ve defined in our controller (identified by $ctrl)



First we need to tie our new function to the vm (add it to our view model/$scope).

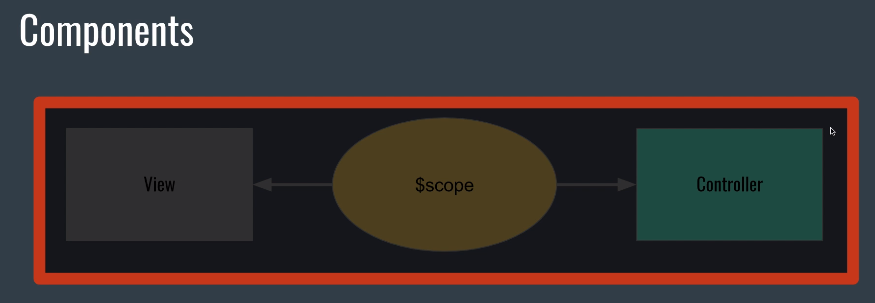


We then define our function. Note that we aren’t returning anything because as soon as we update our model the view will update – that’s :



**Components**

A component is way to gather the view, controller, scope, etc. into a single bundle that has a name. It’s atomic, reusable, so forth.



We can break things up into components now. We’ll declare:

* A module in its own file. Ha

angular

  .module('customFormModule', []) //Note - We're declaring a module, saying it doesn't have any dependencies, and not declaring a controller - we'll do that elsewhere.

  .component('customFormComponent', { // Component called customUiElement

    templateUrl: 'customForm.html', // Points to an HTML template

    controller: 'customFormController' // Defines the controller it uses.

  }); // We're now creating a component. This takes two props: an HTML template of some kind and a controller to tie it to the view model.

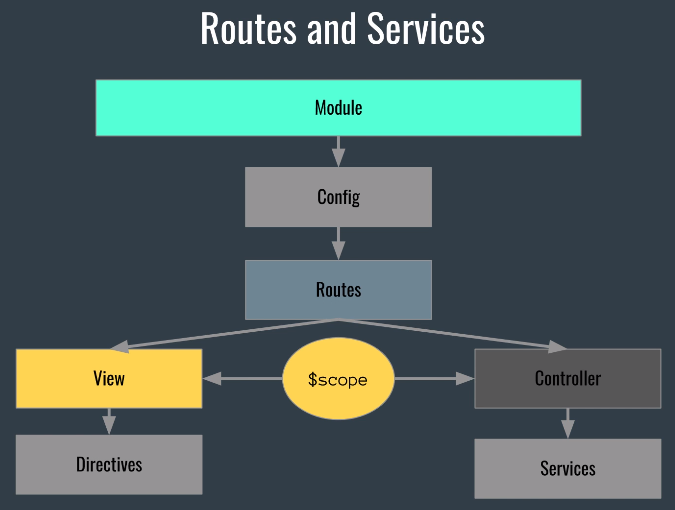
**Routes & Services:**

The hierarchy of an angular app.

Module contains the views, controllers, everything else.

Config is pulled in when you first run. It’s where you define your routes and such when you first start up.

Routes match a URL to a view and a controller. When the URL changes it decides what specific view and or controller we show.



We’re going to do it a bit differently – we’re going to use components, so ‘when the URL changes to this, use this particular component. When it changes again use this other component’.

