**Step 1: Remove/Avoid $scope**

*// before*

angular.module('plunker')

.controller('HomeController', **function**($scope) {

$scope.message = 'Hi from home';

});

*// after*

angular.module('plunker')

.controller('HomeController', **function**() {

**var** vm = this;

vm.message = 'Hi from home';

});

**In router config file:**

*// before*

$stateProvider

.state('home', {

url: '/',

templateUrl: 'home/home.html',

controller: 'HomeController'

})

...

*// after*

$stateProvider

.state('home', {

url: '/',

templateUrl: 'home/home.html',

controller: 'HomeController',

controllerAs: 'vm'

})

...

**In HTML:**

<!-- Before --> <!-- After -->

<h1>Home</h1> <h1>Home</h1>

<p>{{ this.message }}</p> <p>{{ vm.message }}</p>

**Step 2: Convert your controller to a directive**

**Binding controller to view - Before:**

**Option 1(In HTML):**

<div ng-controller="HomeController as vm">... </div>

**Option 2(In the router):**

$stateProvider

.**state**('home', {

url: '/',

templateUrl: 'home/home.html',

controller: 'HomeController',

controllerAs: 'vm'

})

Notice how we define a template and the corresponding controller. This is a very loose coupling. Theoretically we could use one controller for multiple HTML templates easily. That’s considered bad practice, though. Also, this hinders reusability, because one has to know which template and which controller belong together in order to be able to reuse them in another situation.

So let’s change that and stick them together. How? By writing a directive and converting our controller into a directive controller. We have different options:

.directive('home', **function**() {

**return** {

restrict: 'E',

scope: {},

template: 'home/home.html',

controller: HomeController,

controllerAs: 'vm'

}

});

**function** HomeController() {

...

}

**Heads up:** We’re using an isolated scope (**scope: {}** ) for our “**components**”, because we want them to be fully isolated.

Our routing gets simplified, in that it doesn’t have to know the template location and/or the controller, but simply the HTML tag <home>.

$stateProvider

.**state**('home', {

url: '/',

template: '<home></home>'

})

**Step 3: Go further. Use Components!**

*// before*

.directive('home', **function**() {

**return** {

restrict: 'E',

scope: {},

template: 'home/home.html',

controller: HomeController,

controllerAs: 'vm'

}

});

*// after*

.component('home', {

restrict: 'E',

scope: {},

templateUrl: 'home/home.html',

controller: HomeController,

controllerAs: 'vm'

});

We can remove the **controllerAs** property and instead use **$ctrl** property.

*// after*

.component('home', {

restrict: 'E',

scope: {},

templateUrl: 'home/home.html',

controller: HomeController });

…and then in the HTML

<h1>Home</h1>

<p>{{ $ctrl.message }}</p>

**bindToController and bindings:**

When you create directives and pass data into them, you have to define a scope property, right?

.directive('myDirective', {

...

scope: {

message: '='

}

...

})

The problem is that at that point, within your directive controller, you’d have to access them through the $scope variable again, which we previously said should be avoided. Thus, the bindToController has been introduced which allows us to write the above like this:

.directive('myDirective', {

...

scope: {},

bindToController: {

message: '='

}

...

})

Much better, message will now be attached directly to our controller instance. Still, we have to create the **isolate scope** with scope: {} which isn’t the most elegant way of doing it. It gets better . The **new component syntax** simplifies this into a single bindingsproperty, which also creates an isolate scope behind the scenes. (Note we’re no more using  .directive, but instead .component)

.component('myDirective', {

...

bindings: {

message: '='

}

...

})

You can even get **one way bindings**

...

bindings: {

message: '<'

}

...

**Lifecycle hooks:**

.component('home', {

...

controller: HomeController

});

**function** HomeController() {

**var** vm = **this**;

vm.message = '';

activate();

*/////////////////////////*

**function** activate() {

vm.message = 'Hi from home';

}

}

The **activate** method can be seen like the constructor, a place where to group your controller’s initialization code. Well with the new component syntax, you can make use of the **$onInit** hook function.

**function** **HomeController**() {

...

vm.$onInit = activate;

*/////////////////////////*

**function** **activate**() {

vm.message = 'Hi from home';

}

}

There are other hooks, like **$onChange, $onDestroy**and so on.

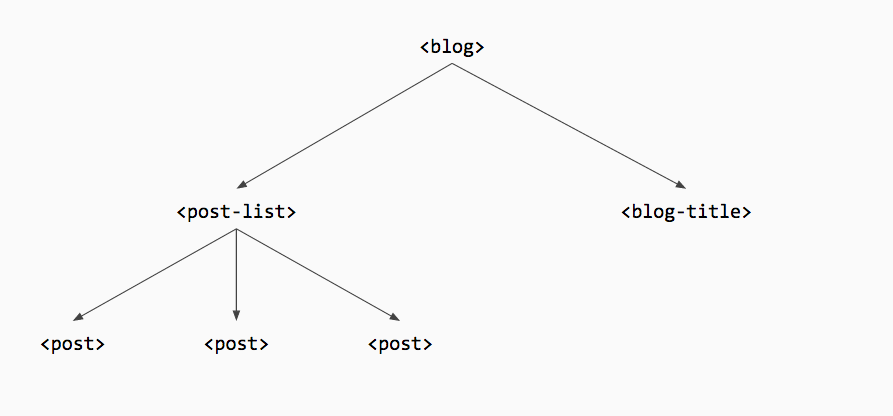
**Component architecture**

What we’ve seen so far are the technical details about how you implement components. It’s however important to also understand the concept behind a component oriented development approach.

The main concept is to define isolated and autonomous components, with a given responsibility and clearly defined contracts in terms of which data flows in and out. Generally speaking, there are two main types of components you usually create, and different people give name them differently:

* **smart components / stateful components** - These are components that coordinate a set of “dumb component”. They connect with Angular services, fetch data or get invoked through routings.
* **dumb components / stateless components** - These are responsible for the immediate visual feedback. They define input bindings and callbacks and render the data they receive. Usually they’re not necessarily coupled to the application and are highly reusable.

What you end with, is a so-called component tree, a set of nested components, starting from a top-level app component or root component.



So a “dumb component” (in every sense) in our sample app could look as follows:

angular.module('plunker')

.component('message', {

bindings: {

**from**: '<',

msg: '<'

},

controller: MessageController,

template: [

'<p><strong>A message from :</strong></p>',

'<p></p>'

].**join**('')

});

function **MessageController**() {}

And be used within our “smart components” <home> and <about> like

<message from="'Home'" msg="$ctrl.message"></message>

**Conclusion:**

What we’ve seen in this article:

* How to refactor $scope to the controllerAs syntax
* How to convert an ng-controller stepwise towards directives and ultimately to components.
* We learned about the new. component syntax introduced in Angular 1.5 and all the benefits we get from it
* How to convert a MV\* pattern like approach into a more component oriented approach.

Summarizing, try to migrate your Angular frontend architecture towards a more component oriented approach. Regardless whether you plan to upgrade to Angular 2 at some point or not, it’ll help you anyway create much cleaner applications.

While this article didn’t deep dive into this topic, but is rather intended to give you a first overview, [Tero Parviainen](https://twitter.com/teropa) has written an in-depth version some time ago: [Refactoring Angular apps to Component Style](http://teropa.info/blog/2015/10/18/refactoring-angular-apps-to-components.html)