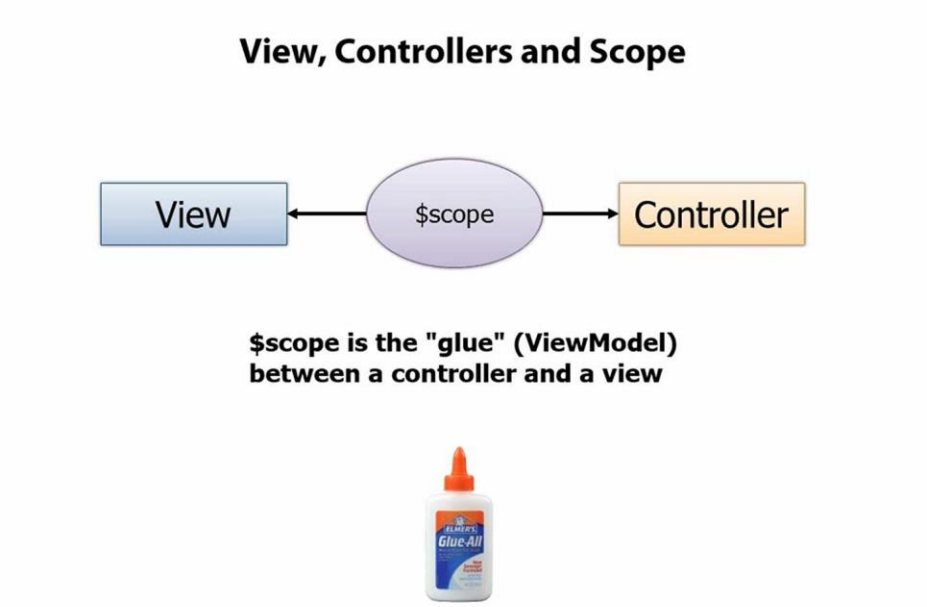
Controllers and Scopes



The way it works in Angular is you have a **View**, which is what we've been doing in the previous section with our **Directives**, our **Filters** and our **Data Binding**.
But we don't want to put all of our logic into the **View** because it's not very maintainable or testable or all those types of things.
Instead we're going to have a special little JavaScript object – a container - called a **Controller**. The **Controller** will drive things. It's going to control ultimately what data gets bound into the **View**.

If the View passes up data to the controller it will handle passing off maybe to a service which then updates a back-end data store.

The glue between the View and the Controller is something called the **Scope**, and in Angular you're going to see a lot of objects or variables that start with $.
$scope represents the scope object.
When I say it's the glue, it literally is the thing that ties the controller to the view.

Now for folks that have worked with **Knockout** or some of the different data binding frameworks out there, not just JavaScript but other desktop modes and things, you might have heard the term "**ViewModel**".

A **ViewModel** literally is the **model** – the data – for the view. Well that's really all the scope is. The scope is our ViewModel and it's the glue between the view and the controller.

1. What are Scopes?

Before we get into Controllers let us understand Scopes. Scope is nothing but an object that Glues the View with Controller. They hold the Model data that we need to pass to view. Scope uses Angular's two-way data binding to bind model data to view.

Imaging $scope as an object that links Controller to the View. It is controllers responsibility to initialize the data that the view needs to display. This is done by making changes to $scope.

Let us see a small Hello World application to understand $scope.

Assigning a Default Value to a Model

Problem

You wish to assign a default value to the scope in the controller's context.

Solution

Use the `ng-controller` directive in your template:

```
<div ng-controller="MyCtrl">
  <p>{{value}}</p>
</div>
```

Next, define the scope variable in your controller function:

```
var MyCtrl = function($scope) {
  $scope.value = "some value";
};
```

Discussion

Depending on where you use the ng-controller directive, you define its assigned scope. The scope is hierarchical and follows the DOM node hierarchy. In our example, the value expression is correctly evaluated to `some value`, since value is set in the `MyCtrl` controller. Note that this would not work if the value expression were moved outside the controllers scope:

```
<p>{{value}}</p>


<div ng-controller="MyCtrl">
</div>
```

In this case {{value}} will simply be not rendered at all due to the fact that expression evaluation in Angular.js is forgiving for `undefined` and `null` values.

Changing a Model Value with a Controller Function

Problem

You wish to increment a model value by 1 using a controller function.

Solution

Implement an increment function that changes the scope.

```
function MyCtrl($scope) {
  $scope.value = 1;


  $scope.incrementValue = function(increment) {
    $scope.value += increment;
  };
}
```

This function can be directly called in an expression, in our example we use `ng-init`:

```
<div ng-controller="MyCtrl">
  <p ng-init="incrementValue(1)">{{value}}</p>
```

```
</div>
```

Discussion

The `ng-init` directive is executed on page load and calls the function `incrementValue` defined in `MyCtrl`. Functions are defined on the scope very similar to values but must be called with the familiar parenthesis syntax.

Of course, it would have been possible to increment the value right inside of the expression with `value = value +1` but imagine the function being much more complex! Moving this function into a controller separates our business logic from our declarative view template and we can easily write unit tests for it.

Encapsulating a Model Value with a Controller Function

Problem

You wish to retrieve a model via function (instead of directly accessing the scope from the template) that encapsulates the model value.

Solution

Define a getter function that returns the model value.

```
function MyCtrl($scope) {
  $scope.value = 1;


  $scope.getIncrementedValue = function() {
    return $scope.value + 1;
  };
}
```

Then in the template we use an expression to call it:

```
<div ng-controller="MyCtrl">
  <p>{{getIncrementedValue()}}</p>
```

```
</div>
```

Discussion

`MyCtrl` defines the `getIncrementedValue` function, which uses the current value and returns it incremented by one. One could argue that depending on the use case it would make more sense to use a filter. But there are use cases specific to the controllers behavior where a generic filter is not required.

Responding to Scope Changes

Problem

You wish to react on a model change to trigger some further actions. In our example we simple want to set another model value depending on the value we are listening to.

Solution

Use the `$watch` function in your controller.

```
function MyCtrl($scope) {
  $scope.name = "";

  $scope.$watch("name", function(newValue, oldValue) {
    if ($scope.name.length > 0) {
      $scope.greeting = "Greetings " + $scope.name;
    }
  });
}
```

In our example we use the text input value to print a friendly greeting.

```
<div ng-controller="MyCtrl">
  <input type="text" ng-model="name" placeholder="Enter your name">
  <p>{{greeting}}</p>
</div>
```

The value `greeting` will be changed whenever there's a change to the `name` model and the value is not blank.

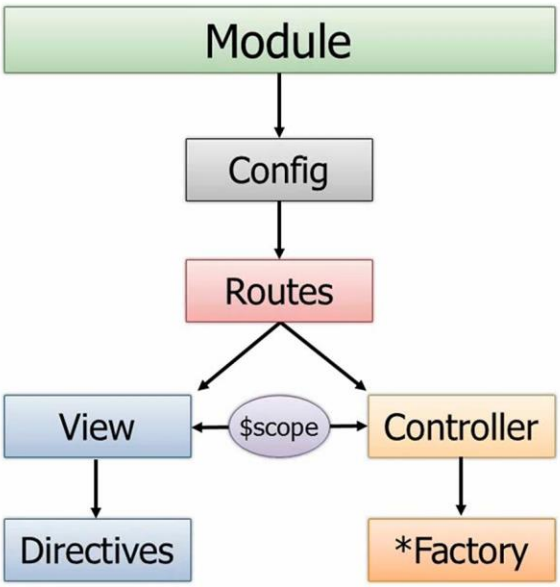
Discussion

The first argument `name` of the `$watch` function is actually an Angular expression, so you can use more complex expressions (for example: `[value1, value2] | json`) or even a Javascript function. In this case you need to return a string in the watcher function:

```
$scope.$watch(function() {
  return $scope.name;
}, function(newValue, oldValue) {
  console.log("change detected: " + newValue)
});
```

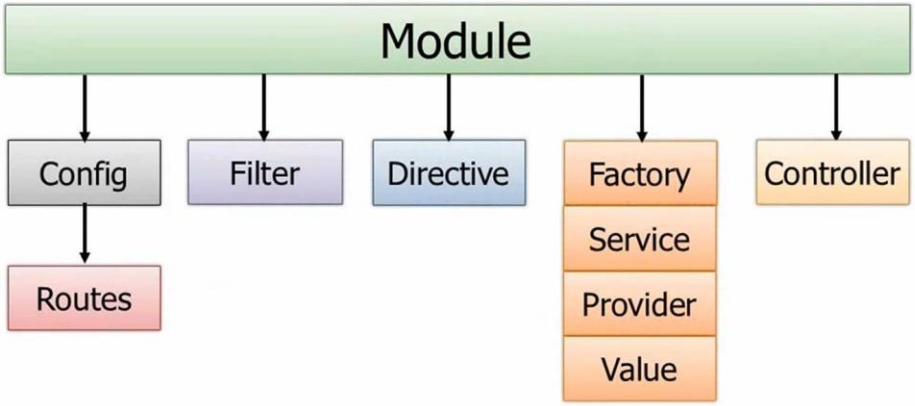
The second argument is a function which gets called whenever the expression evaluation returns a different value. The first parameter is the new value and the second parameter the old value. Internally, this uses `angular.equals` to determine equality which means both objects or values pass the `===` comparison.

Modules, Routes and Factories

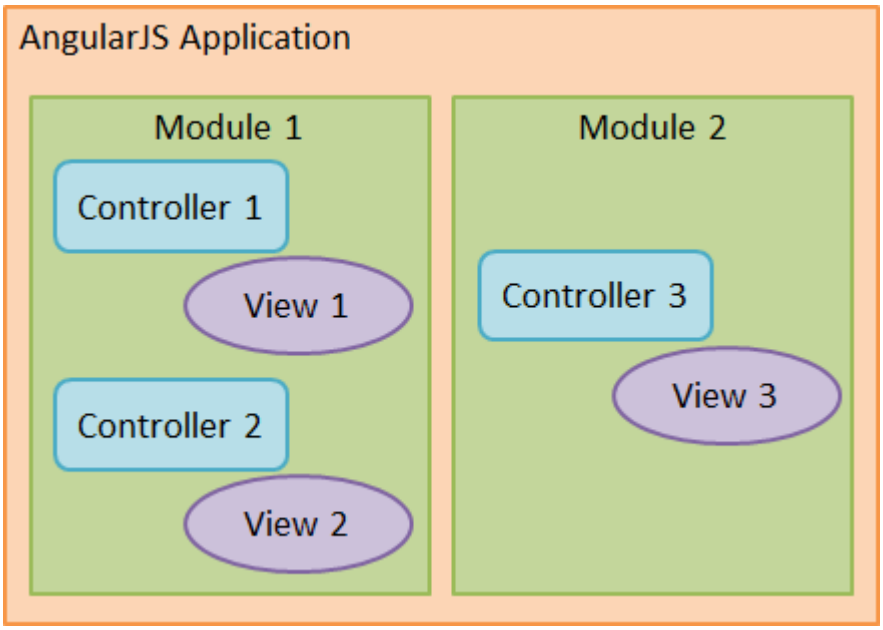


Modules are Containers

```
<html ng-app="moduleName">
```



You have probably noticed that all the examples presented so far were using global constructor functions to define controllers. But global state is evil, it hurts application structure, makes code hard to maintain, test, and read. By no means is AngularJS suggesting usage of global state. On the contrary, it comes with a set of APIs that make it very easy to define modules and register objects in those modules.



Furthermore, each modules have several Controllers. As referred in above diagram, a module can contain one or more controllers and views. We haven't touch based on Views. We will see how each module can be linked with view using Routing in AngularJS. For now just assume that each controller has one or more views linked to it.

Modules in AngularJS

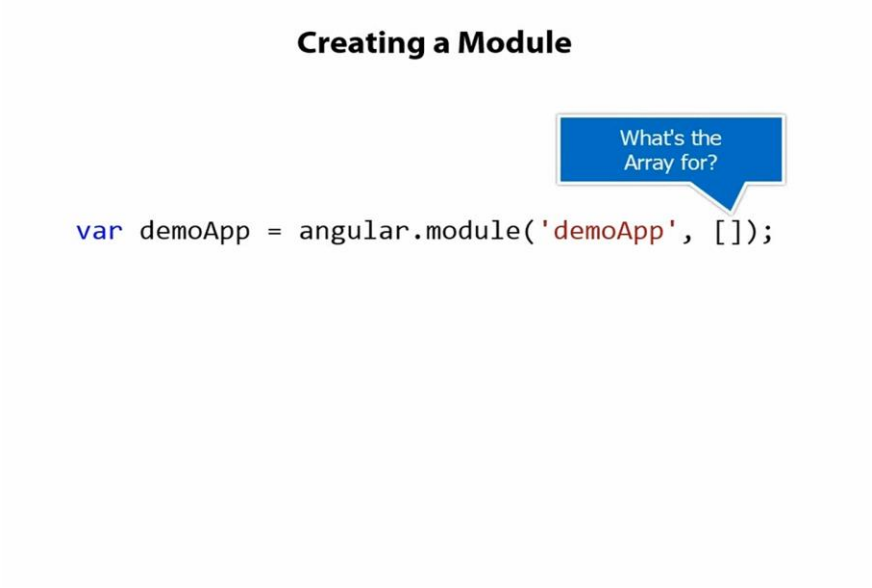
Let's see how to turn an ugly, globally-defined controller definition into its modular equivalent, before a controller is declared as follows:

```
var HelloCtrl = function ($scope) {
$scope.name = 'World';
}
```

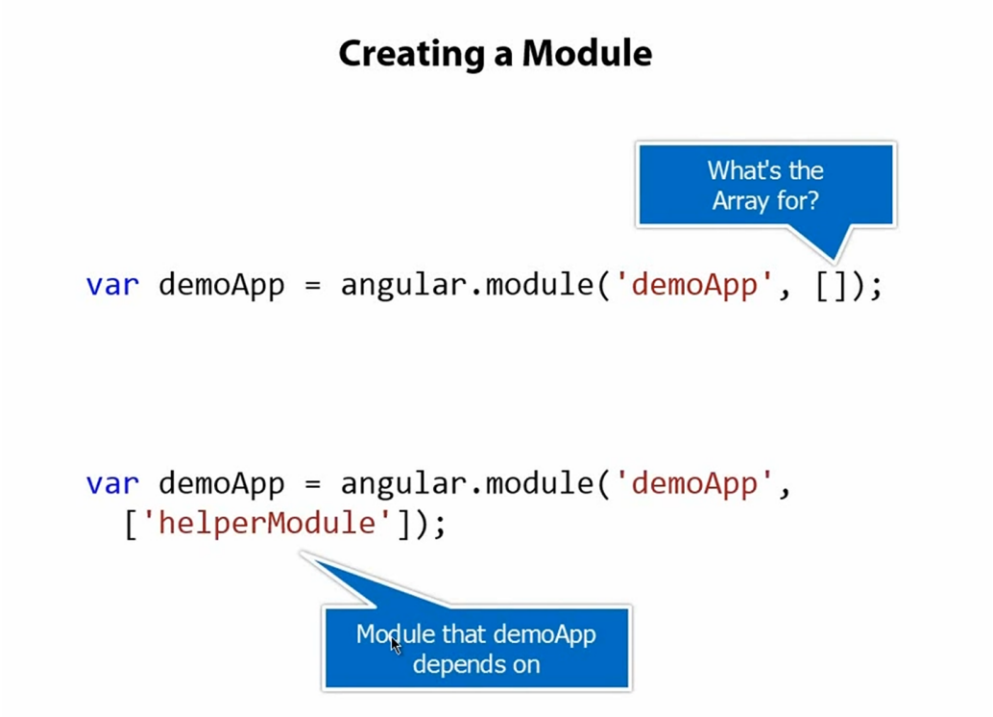
And when using modules it looks as follows:

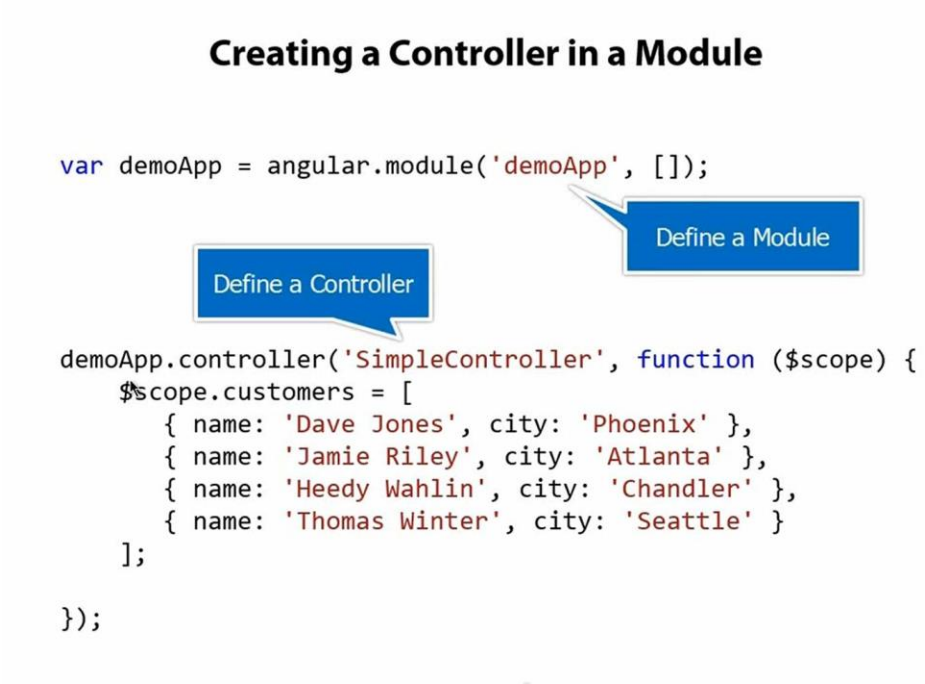
```
angular.module('hello', [])
.controller('HelloCtrl', function($scope){
```

```
$scope.name = 'World';
});
```



In this example you can see I called my module **demoApp**. You might wonder what exactly is the empty array for? I know I did the first time I saw it.
The answer is this is where **dependency injection** comes in because your module might actually rely on other modules to get data.





Multiple Controller

We can define multiple controllers in the same HTML template and each one manages its own $scope object.

In the example we have two controllers, firstController and secondController, and even though there are some model variables named in the same way for both the controllers, each one resides in its own $scope thus avoiding any conflict (they are in fact different variables even though the name is the same).

Example

Nested Controllers

In demo, notice how each nested Controller's scope override the scope of parents controller. First we defined a controller `CarController` which defines two variables `name` and `type` within scope. Next `BMWController` is nested within CarController using `ng-controller` attribute. BMWController overrides `name` attribute and change it to BMW. It does not change `type` attribute so type attribute is still Car.

`BMWMotorcycleController` is the inner-most controller defined within controllers hierarchy. It overrides both `name` and `type` attribute of scope.

This way you can nest one controller within another and take advantage of parent controllers attributes whenever needed.

Example

Inheritance in Controllers

In order to take advantage of inheritance of scope in Nested controllers, one has to define Controllers one into another using `ng-controller` attribute. Sometimes you don't want to define controllers like this but still want to use power of inheritance within controllers. May be you want to put common logic into BaseController and use it in all the child controllers.
In order to achieve this, we must use `$injector` object that AngularJS provides.

Example

Nested Controller

Scopes in depth

Each $scope is an instance of the Scope class. The Scope class has methods that control the scope's lifecycle, provide event-propagation facility, and support the template rendering process.

A new scope was created by the ng-controller directive using the Scope.$new() method call.

The $rootScope instance gets created when a new application is bootstrapped. (a scope that is a parent of all the other scopes).

Scopes hierarchy and inheritance

Properties defined on one scope are visible to all child scopes, provided that a child scope doesn't redefine a property using the same name! This is very useful in practice, since we don't need to redefine over and over again properties that should be available throughout a scope hierarchy.

Building on our previous example, let's assume that we want to display the percentage
of the world's population that lives in a given country. To do so, we can define the
worldsPercentage function on a scope managed by the WorldCtrl as given in the
following code:

```
$scope.worldsPercentage = function (countryPopulation) {
return (countryPopulation / $scope.population)*100;
}
And then call this function from each scope instance created by the ng-repeat
directive as follows:
<li ng-repeat="country in countries">
{{country.name}} has population of {{country.population}},
{{worldsPercentage(country.population)}} % of the World's
population
</li>
```

Scope's inheritance in AngularJS follows the same rules as prototypical inheritance
in JavaScript (when we try to read a property, the inheritance tree will be traversed
upwards till a property is found).

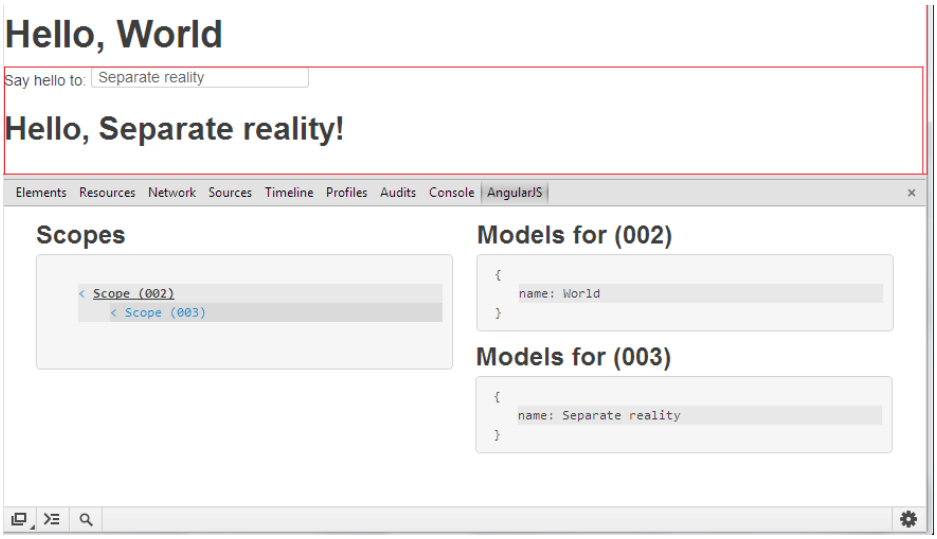
Perils of the inheritance through the scopes hierarchy

Inheritance through the scopes hierarchy is rather intuitive and easy to understand when it comes to the read access. When it comes to the write access, however, things become a little bit complicated.

```
var HelloCtrl = function ($scope) {
};
```

And the view code is as follows:

```
<body ng-app ng-init="name='World'">
<h1>Hello, {{name}}</h1>
<div ng-controller="HelloCtrl">
Say hello to: <input type="text" ng-model="name">
<h2>Hello, {{name}}!</h2>
```



You might be a bit surprised to see that a new variable was created in the scope initialized by the HelloCtrl controller, instead of changing a value set up on the $rootScope instance. This behavior becomes less surprising when we realize that scopes prototypically inherit from each other. All the rules that apply to the

prototypical inheritance of objects in JavaScript apply equally to scopes prototypical inheritance. Scopes are just JavaScript objects after all.

Scope Life Cycle

- **Creation**

The root scope is created during the application bootstrap by the $injector. During template linking, some directives create new child scopes.

- **Watcher registration**

During template linking directives register watches on the scope. These watches will be used to propagate model values to the DOM.

- **Model mutation**

For mutations to be properly observed, you should make them only within the scope.$apply(). (Angular APIs do this implicitly, so no extra `$apply` call is needed when doing synchronous work in controllers, or asynchronous work with $http, $timeout or $intervalservices.

- **Mutation observation**

At the end of `$apply`, Angular performs a $digest cycle on the root scope, which then propagates throughout all child scopes. During the `$digest` cycle, all `$watch`ed expressions or functions are checked for model mutation and if a mutation is detected, the`$watch` listener is called.

- **Scope destruction**

When child scopes are no longer needed, it is the responsibility of the child scope creator to destroy them via scope.$destroy() API. This will stop propagation of `$digest` calls into the child scope and allow for memory used by the child scope models to be reclaimed by the garbage collector.
