## Section 3

# Controllers

# Understanding Controllers

In Angular, a Controller is a JavaScript **constructor function** that is used to augment the [Angular Scope](https://docs.angularjs.org/guide/scope).

When a Controller is attached to the DOM via the [ng-controller](https://docs.angularjs.org/api/ng/directive/ngController) directive, Angular will instantiate a new Controller object, using the specified Controller's **constructor function**. A new **child scope** will be available as an injectable parameter to the Controller's constructor function as $scope.

Use controllers to:

* Set up the initial state of the $scope object.
* Add behavior to the $scope object.

Do not use controllers to:

* Manipulate DOM — Controllers should contain only business logic. Putting any presentation logic into Controllers significantly affects its testability. Angular has [databinding](https://docs.angularjs.org/guide/databinding) for most cases and[directives](https://docs.angularjs.org/guide/directive) to encapsulate manual DOM manipulation.
* Format input — Use [angular form controls](https://docs.angularjs.org/guide/forms) instead.
* Filter output — Use [angular filters](https://docs.angularjs.org/guide/filter) instead.
* Share code or state across controllers — Use [angular services](https://docs.angularjs.org/guide/services) instead.
* Manage the life-cycle of other components (for example, to create service instances).

# Setting up the initial state of a *$scope* object

Typically, when you create an application you need to set up the initial state for the Angular $scope. You set up the initial state of a scope by attaching properties to the $scope object. The properties contain the**view model** (the model that will be presented by the view). All the $scope properties will be available to the template at the point in the DOM where the Controller is registered.

The following example demonstrates creating a GreetingController, which attaches a greetingproperty containing the string 'Hola!' to the $scope:

var myApp = angular.module('myApp',[]);

myApp.controller('GreetingController', ['$scope', function($scope) {

$scope.greeting = 'Hola!';

}]);

We create an [Angular Module](https://docs.angularjs.org/guide/module), myApp, for our application. Then we add the controller's constructor function to the module using the .controller() method. This keeps the controller's constructor function out of the global scope.

We have used an **inline injection annotation** to explicitly specify the dependency of the Controller on the $scope service provided by Angular. See the guide on [Dependency Injection](https://docs.angularjs.org/guide/di) for more information.

We attach our controller to the DOM using the ng-controller directive. The greeting property can now be data-bound to the template:

<div ng-controller="GreetingController">

{{ greeting }}

</div>

# Adding Behavior to a Scope Object

In order to react to events or execute computation in the view we must provide behavior to the scope. We add behavior to the scope by attaching methods to the $scope object. These methods are then available to be called from the template/view.

The following example uses a Controller to add a method to the scope, which doubles a number:

var myApp = angular.module('myApp',[]);

myApp.controller('DoubleController', ['$scope', function($scope) {

$scope.double = function(value) { return value \* 2; };

}]);

Once the Controller has been attached to the DOM, the double method can be invoked in an Angular expression in the template:

<div ng-controller="DoubleController">

Two times <input ng-model="num"> equals {{ double(num) }}

</div>

As discussed in the [Concepts](https://docs.angularjs.org/guide/concepts) section of this guide, any objects (or primitives) assigned to the scope become model properties. Any methods assigned to the scope are available in the template/view, and can be invoked via angular expressions and ng event handler directives (e.g. [ngClick](https://docs.angularjs.org/api/ng/directive/ngClick)).

# Using Controllers Correctly

In general, a Controller shouldn't try to do too much. It should contain only the business logic needed for a single view.

The most common way to keep Controllers slim is by encapsulating work that doesn't belong to controllers into services and then using these services in Controllers via dependency injection. This is discussed in the [Dependency Injection](https://docs.angularjs.org/guide/di) [Services](https://docs.angularjs.org/guide/services) sections of this guide.

# Associating Controllers with Angular Scope Objects

You can associate Controllers with scope objects implicitly via the [ngController directive](https://docs.angularjs.org/api/ng/directive/ngController) or [$route service](https://docs.angularjs.org/api/ngRoute/service/$route).

## Simple Spicy Controller Example

To illustrate further how Controller components work in Angular, let's create a little app with the following components:

* A [template](https://docs.angularjs.org/guide/templates) with two buttons and a simple message
* A model consisting of a string named spice
* A Controller with two functions that set the value of spice

The message in our template contains a binding to the spice model, which by default is set to the string "very". Depending on which button is clicked, the spice model is set to chili or jalapeño, and the message is automatically updated by data-binding.

Code: test\_5.html, test\_5.js

Things to notice in the example above:

* The ng-controller directive is used to (implicitly) create a scope for our template, and the scope is augmented (managed) by the SpicyController Controller.
* SpicyController is just a plain JavaScript function. As an (optional) naming convention the name starts with capital letter and ends with "Controller".
* Assigning a property to $scope creates or updates the model.
* Controller methods can be created through direct assignment to scope (see the chiliSpicymethod)
* The Controller methods and properties are available in the template (for the <div> element and its children).

## Spicy Arguments Example

Controller methods can also take arguments, as demonstrated in the following variation of the previous example.

Code: test\_6.html, test\_6.js

Notice that the SpicyController Controller now defines just one method called spicy, which takes one argument called spice. The template then refers to this Controller method and passes in a string constant 'chili' in the binding for the first button and a model property customSpice (bound to an input box) in the second button.

## Scope Inheritance Example

It is common to attach Controllers at different levels of the DOM hierarchy. Since the [ng-controller](https://docs.angularjs.org/api/ng/directive/ngController) directive creates a new child scope, we get a hierarchy of scopes that inherit from each other. The $scope that each Controller receives will have access to properties and methods defined by Controllers higher up the hierarchy. See [Understanding Scopes](https://github.com/angular/angular.js/wiki/Understanding-Scopes) for more information about scope inheritance.

Code: test\_7.html, test\_7.js

Notice how we nested three ng-controller directives in our template. This will result in four scopes being created for our view:

* The root scope
* The MainController scope, which contains timeOfDay and name properties
* The ChildController scope, which inherits the timeOfDay property but overrides (hides) thename property from the previous
* The GrandChildController scope, which overrides (hides) both the timeOfDay property defined inMainController and the name property defined in ChildController

Inheritance works with methods in the same way as it does with properties. So in our previous examples, all of the properties could be replaced with methods that return string values.