AngularJS API Docs

https://docs.angularjs.org/api

Welcome to the AngularJS API docs page. These pages contain the AngularJS reference materials for version **1.4.0-rc.0 smooth-unwinding**.

The documentation is organized into **modules** which contain various components of an AngularJS application. These components are directives, services, filters, providers, templates, global APIs, and testing mocks.

Angular Prefixes `$` and `$$`: To prevent accidental name collisions with your code, Angular prefixes names of public objects with `$` and names of private objects with `$$`. Please do not use the `$` or `$$` prefix in your code.

Angular Modules

ng (core module)

This module is provided by default and contains the core components of AngularJS.

Directives	This is the core collection of directives you would use in your template code to build an AngularJS application. Some examples include: ngClick, ngInclude, ngRepeat, etc…
Services / Factories	This is the core collection of services which are used within the DI of your application. Some examples include: $compile, $http, $location, etc…
Filters	The core filters available in the ng module are used to transform template data before it is rendered within directives and expressions. Some examples include: filter, date, currency, lowercase, uppercase, etc...
Global APIs	The core global API functions are attached to the angular object. These core functions are useful for low level JavaScript operations within your application. Some examples include: angular.copy(), angular.equals(), angular.element(), etc...

ngRoute

Use ngRoute to enable URL routing to your application. The ngRoute module supports URL management via both hashbang and HTML5 pushState.

Include the **angular-route.js** file and set **ngRoute** as a dependency for this to work in your application.

Services / Factories	The following services are used for route management: • $routeParams is used to access the querystring values present in the URL. • $route is used to access the details of the route that is currently being accessed. • $routeProvider is used to register routes for the application.
Directives	The ngView directive will display the template of the current route within the page.

ngAnimate

Use ngAnimate to enable animation features within your application. Various core ng directives will provide animation hooks into your application when ngAnimate is included. Animations are defined by using CSS transitions/animations or JavaScript callbacks.

Include the **angular-animate.js** file and set **ngAnimate** as a dependency for this to work in your application.

Services / Factories	Use $animate to trigger animation operations within your directive code.
CSS-based animations	Follow ngAnimate's CSS naming structure to reference CSS transitions / keyframe animations in AngularJS. Once defined, the animation can be triggered by referencing the CSS class within the HTML template code.
JS-based animations	Use module.animation() to register a JavaScript animation. Once registered, the animation can be triggered by referencing the CSS class within the HTML template code.

ngAria

Use ngAria to inject common accessibility attributes into directives and improve the experience for users with disabilities.

Include the **angular-aria.js** file and set ngAria as a dependency for this to work in your application.

Services	The $aria service contains helper methods for applying ARIA attributes to HTML. $ariaProvider is used for configuring ARIA attributes.

ngResource

Use the ngResource module when querying and posting data to a REST API.

Include the **angular-resource.js** file and set **ngResource** as a dependency for this to work in your application.

Services / Factories	The $resource service is used to define RESTful objects which communicate with a REST API.

ngCookies

Use the ngCookies module to handle cookie management within your application.

Include the **angular-cookies.js** file and set **ngCookies** as a dependency for this to work in your application.

Services / Factories	The following services are used for cookie management: • The $cookie service is a convenient wrapper to store simple data within browser cookies. • $cookieStore is used to store more complex data using serialization.

ngTouch

Use ngTouch when developing for mobile browsers/devices.

Include the **angular-touch.js** file and set **ngTouch** as a dependency for this to work in your application.

Services / Factories	The $swipe service is used to register and manage mobile DOM events.
Directives	Various directives are available in ngTouch to emulate mobile DOM events.

ngSanitize

Use ngSanitize to securely parse and manipulate HTML data in your application.

Include the **angular-sanitize.js** file and set **ngSanitize** as a dependency for this to work in your application.

Services / Factories	The $sanitize service is used to clean up dangerous HTML code in a quick and convenient way.

Filters	The linky filter is used to turn URLs into HTML links within the provided string.

ngMock

Use ngMock to inject and mock modules, factories, services and providers within your unit tests.

Include the **angular-mocks.js** file into your test runner for this to work.

Services / Factories	ngMock will extend the behavior of various core services to become testing aware and manageable in a synchronous manner. Some examples include: $timeout, $interval, $log, $httpBackend, etc...
Global APIs	Various helper functions are available to inject and mock modules within unit test code. Some examples inject(), module(), dump(), etc...

ng

ng (core module)

The ng module is loaded by default when an AngularJS application is started. The module itself contains the essential components for an AngularJS application to function. The table below lists a high level breakdown of each of the services/factories, filters, directives and testing components available within this core module.

Module Components

Function

Name	Description
angular.lowercase	Converts the specified string to lowercase.
angular.uppercase	Converts the specified string to uppercase.
angular.forEach	Invokes the `iterator` function once for each item in `obj` collection, which can be either an object or an array. The `iterator` function is invoked with `iterator(value, key, obj)`, where `value` is the value of an object property or an array element, `key` is the object property key or array element index and obj is the `obj` itself. Specifying a `context` for the function is optional.
angular.extend	Extends the destination object `dst` by copying own enumerable properties from the `src` object(s) to `dst`. You can specify multiple `src` objects. If you want to preserve original objects, you can do so by passing an empty object as the target: `var object = angular.extend({}, object1, object2)`.
angular.merge	Deeply extends the destination object `dst` by copying own enumerable properties from the `src` object(s) to `dst`. You can specify multiple `src` objects. If you want to preserve original objects, you can do so by passing an empty object as the target: `var object = angular.merge({}, object1, object2)`.
angular.noop	A function that performs no operations. This function can be useful when writing code in the functional style.

	`function foo(callback) { var result = calculateResult(); (callback || angular.noop)(result); }`
angular.identity	A function that returns its first argument. This function is useful when writing code in the functional style.
angular.isUndefined	Determines if a reference is undefined.
angular.isDefined	Determines if a reference is defined.
angular.isObject	Determines if a reference is an `Object`. Unlike `typeof` in JavaScript, `null`s are not considered to be objects. Note that JavaScript arrays are objects.
angular.isString	Determines if a reference is a `String`.
angular.isNumber	Determines if a reference is a `Number`.
angular.isDate	Determines if a value is a date.
angular.isArray	Determines if a reference is an `Array`.
angular.isFunction	Determines if a reference is a `Function`.
angular.isElement	Determines if a reference is a DOM element (or wrapped jQuery element).
angular.copy	Creates a deep copy of `source`, which should be an object or an array.
angular.equals	Determines if two objects or two values are equivalent. Supports value types, regular expressions, arrays and objects.
angular.bind	Returns a function which calls function `fn` bound to `self` (`self` becomes the `this` for `fn`). You can supply optional `args` that are prebound to the function. This feature is also known as partial application, as distinguished from function currying.
angular.toJson	Serializes input into a JSON-formatted string. Properties with leading $$ characters will be stripped since angular uses this notation internally.
angular.fromJson	Deserializes a JSON string.
angular.bootstrap	Use this function to manually start up angular application.
angular.reloadWithDebugInfo	Use this function to reload the current application with debug information turned on. This takes precedence over a call to `$compileProvider.debugInfoEnabled(false)`.
angular.injector	Creates an injector object that can be used for retrieving services as well as for dependency injection (see dependency injection).

angular.element	Wraps a raw DOM element or HTML string as a jQuery element.
angular.module	The `angular.module` is a global place for creating, registering and retrieving Angular modules. All modules (angular core or 3rd party) that should be available to an application must be registered using this mechanism.

Directive

Name	Description
ngJq	Use this directive to force the angular.element library. This should be used to force either jqLite by leaving ng-jq blank or setting the name of the jquery variable under window (eg. jQuery).
ngApp	Use this directive to **auto-bootstrap** an AngularJS application. The `ngApp` directive designates the **root element** of the application and is typically placed near the root element of the page - e.g. on the `<body>` or `<html>` tags.
a	Modifies the default behavior of the html A tag so that the default action is prevented when the href attribute is empty.
ngHref	Using Angular markup like `{{hash}}` in an href attribute will make the link go to the wrong URL if the user clicks it before Angular has a chance to replace the `{{hash}}` markup with its value. Until Angular replaces the markup the link will be broken and will most likely return a 404 error. The `ngHref` directive solves this problem.
ngSrc	Using Angular markup like `{{hash}}` in a `src` attribute doesn't work right: The browser will fetch from the URL with the literal text `{{hash}}` until Angular replaces the expression inside `{{hash}}`. The `ngSrc` directive solves this problem.
ngSrcset	Using Angular markup like `{{hash}}` in a `srcset` attribute doesn't work right: The browser will fetch from the URL with the literal text `{{hash}}` until Angular replaces the expression inside `{{hash}}`. The `ngSrcset` directive solves this problem.
ngDisabled	This directive sets the `disabled` attribute on the element if the expression inside `ngDisabled` evaluates to truthy.
ngChecked	The HTML specification does not require browsers to preserve the values of boolean attributes such as checked. (Their presence means true and their absence means false.) If we put an Angular interpolation expression into such an attribute then the binding information would be lost when the browser removes the attribute. The `ngChecked` directive solves this problem for the `checked` attribute. This complementary directive is not removed by the browser and so provides a permanent reliable place to store the binding information.
ngReadonly	The HTML specification does not require browsers to preserve the values of boolean attributes such as readonly. (Their presence means true and their absence means false.) If we put an Angular interpolation expression into such an attribute then the binding information would be lost when the browser removes the attribute. The `ngReadonly` directive solves this problem for the

	`readonly` attribute. This complementary directive is not removed by the browser and so provides a permanent reliable place to store the binding information.
ngSelected	The HTML specification does not require browsers to preserve the values of boolean attributes such as selected. (Their presence means true and their absence means false.) If we put an Angular interpolation expression into such an attribute then the binding information would be lost when the browser removes the attribute. The `ngSelected` directive solves this problem for the `selected` attribute. This complementary directive is not removed by the browser and so provides a permanent reliable place to store the binding information.
ngOpen	The HTML specification does not require browsers to preserve the values of boolean attributes such as open. (Their presence means true and their absence means false.) If we put an Angular interpolation expression into such an attribute then the binding information would be lost when the browser removes the attribute. The `ngOpen` directive solves this problem for the `open` attribute. This complementary directive is not removed by the browser and so provides a permanent reliable place to store the binding information.
ngForm	Nestable alias of `form` directive. HTML does not allow nesting of form elements. It is useful to nest forms, for example if the validity of a sub-group of controls needs to be determined.
form	Directive that instantiates FormController.
textarea	HTML textarea element control with angular data-binding. The data-binding and validation properties of this element are exactly the same as those of the input element.
input	HTML input element control. When used together with `ngModel`, it provides data-binding, input state control, and validation. Input control follows HTML5 input types and polyfills the HTML5 validation behavior for older browsers.
ngValue	Binds the given expression to the value of `<option>` or `input[radio]`, so that when the element is selected, the `ngModel` of that element is set to the bound value.
ngBind	The `ngBind` attribute tells Angular to replace the text content of the specified HTML element with the value of a given expression, and to update the text content when the value of that expression changes.
ngBindTemplate	The `ngBindTemplate` directive specifies that the element text content should be replaced with the interpolation of the template in the `ngBindTemplate` attribute. Unlike `ngBind`, the `ngBindTemplate` can contain multiple `{{ }}` expressions. This directive is needed since some HTML elements (such as TITLE and OPTION) cannot contain SPAN elements.
ngBindHtml	Evaluates the expression and inserts the resulting HTML into the element in a secure way. By default, the resulting HTML content will be sanitized using the $sanitize service. To utilize this functionality, ensure that `$sanitize` is available, for example, by including `ngSanitize` in your module's dependencies (not in core Angular). In order to use `ngSanitize` in your module's dependencies, you need to include "angular-sanitize.js" in your application.

ngChange	Evaluate the given expression when the user changes the input. The expression is evaluated immediately, unlike the JavaScript onchange event which only triggers at the end of a change (usually, when the user leaves the form element or presses the return key).
ngClass	The `ngClass` directive allows you to dynamically set CSS classes on an HTML element by databinding an expression that represents all classes to be added.
ngClassOdd	The `ngClassOdd` and `ngClassEven` directives work exactly as ngClass, except they work in conjunction with `ngRepeat` and take effect only on odd (even) rows.
ngClassEven	The `ngClassOdd` and `ngClassEven` directives work exactly as ngClass, except they work in conjunction with `ngRepeat` and take effect only on odd (even) rows.
ngCloak	The `ngCloak` directive is used to prevent the Angular html template from being briefly displayed by the browser in its raw (uncompiled) form while your application is loading. Use this directive to avoid the undesirable flicker effect caused by the html template display.
ngController	The `ngController` directive attaches a controller class to the view. This is a key aspect of how angular supports the principles behind the Model-View-Controller design pattern.
ngCsp	Enables CSP (Content Security Policy) support.
ngClick	The ngClick directive allows you to specify custom behavior when an element is clicked.
ngDblclick	The `ngDblclick` directive allows you to specify custom behavior on a dblclick event.
ngMousedown	The ngMousedown directive allows you to specify custom behavior on mousedown event.
ngMouseup	Specify custom behavior on mouseup event.
ngMouseover	Specify custom behavior on mouseover event.
ngMouseenter	Specify custom behavior on mouseenter event.
ngMouseleave	Specify custom behavior on mouseleave event.
ngMousemove	Specify custom behavior on mousemove event.
ngKeydown	Specify custom behavior on keydown event.
ngKeyup	Specify custom behavior on keyup event.
ngKeypress	Specify custom behavior on keypress event.
ngSubmit	Enables binding angular expressions to onsubmit events.
ngFocus	Specify custom behavior on focus event.

ngBlur	Specify custom behavior on blur event.
ngCopy	Specify custom behavior on copy event.
ngCut	Specify custom behavior on cut event.
ngPaste	Specify custom behavior on paste event.
ngIf	The `ngIf` directive removes or recreates a portion of the DOM tree based on an {expression}. If the expression assigned to `ngIf` evaluates to a false value then the element is removed from the DOM, otherwise a clone of the element is reinserted into the DOM.
ngInclude	Fetches, compiles and includes an external HTML fragment.
ngInit	The `ngInit` directive allows you to evaluate an expression in the current scope.
ngList	Text input that converts between a delimited string and an array of strings. The default delimiter is a comma followed by a space - equivalent to `ng-list=", "`. You can specify a custom delimiter as the value of the `ngList` attribute - for example, `ng-list=" | "`.
ngModel	The `ngModel` directive binds an `input`,`select`, `textarea` (or custom form control) to a property on the scope using NgModelController, which is created and exposed by this directive.
ngModelOptions	Allows tuning how model updates are done. Using `ngModelOptions` you can specify a custom list of events that will trigger a model update and/or a debouncing delay so that the actual update only takes place when a timer expires; this timer will be reset after another change takes place.
ngNonBindable	The `ngNonBindable` directive tells Angular not to compile or bind the contents of the current DOM element. This is useful if the element contains what appears to be Angular directives and bindings but which should be ignored by Angular. This could be the case if you have a site that displays snippets of code, for instance.
ngOptions	The `ngOptions` attribute can be used to dynamically generate a list of `<option>` elements for the `<select>` element using the array or object obtained by evaluating the `ngOptions` comprehension expression.
ngPluralize	`ngPluralize` is a directive that displays messages according to en-US localization rules. These rules are bundled with angular.js, but can be overridden (see Angular i18n dev guide). You configure ngPluralize directive by specifying the mappings between plural categories and the strings to be displayed.
ngRepeat	The `ngRepeat` directive instantiates a template once per item from a collection. Each template instance gets its own scope, where the given loop variable is set to the current collection item, and `$index` is set to the item index or key.
ngShow	The `ngShow` directive shows or hides the given HTML element based on the expression provided to the `ngShow` attribute. The element is shown or hidden by removing or adding the `.ng-hide` CSS class onto the element. The `.ng-hide`

	CSS class is predefined in AngularJS and sets the display style to none (using an !important flag). For CSP mode please add `angular-csp.css` to your html file (see ngCsp).
ngHide	The `ngHide` directive shows or hides the given HTML element based on the expression provided to the `ngHide` attribute. The element is shown or hidden by removing or adding the `ng-hide` CSS class onto the element. The `.ng-hide` CSS class is predefined in AngularJS and sets the display style to none (using an !important flag). For CSP mode please add `angular-csp.css` to your html file (see ngCsp).
ngStyle	The `ngStyle` directive allows you to set CSS style on an HTML element conditionally.
ngSwitch	The `ngSwitch` directive is used to conditionally swap DOM structure on your template based on a scope expression. Elements within `ngSwitch` but without `ngSwitchWhen` or `ngSwitchDefault` directives will be preserved at the location as specified in the template.
ngTransclude	Directive that marks the insertion point for the transcluded DOM of the nearest parent directive that uses transclusion.
script	Load the content of a `<script>` element into `$templateCache`, so that the template can be used by `ngInclude`, `ngView`, or directives. The type of the `<script>` element must be specified as `text/ng-template`, and a cache name for the template must be assigned through the element's `id`, which can then be used as a directive's `templateUrl`.
select	HTML `SELECT` element with angular data-binding.

Object

Name	Description
angular.version	An object that contains information about the current AngularJS version. This object has the following properties:

Type

Name	Description
angular.Module	Interface for configuring angular modules.
$cacheFactory.Cache	A cache object used to store and retrieve data, primarily used by $http and the script directive to cache templates and other data.
$compile.directive.Attributes	A shared object between directive compile / linking functions which contains normalized DOM element attributes. The values reflect current binding state `{{ }}`. The normalization is needed since all of these are treated as equivalent in Angular:

form.FormController	`FormController` keeps track of all its controls and nested forms as well as the state of them, such as being valid/invalid or dirty/pristine.
ngModel.NgModelController	`NgModelController` provides API for the `ngModel` directive. The controller contains services for data-binding, validation, CSS updates, and value formatting and parsing. It purposefully does not contain any logic which deals with DOM rendering or listening to DOM events. Such DOM related logic should be provided by other directives which make use of `NgModelController` for data-binding to control elements. Angular provides this DOM logic for most `input` elements. At the end of this page you can find a custom control example that uses `ngModelController` to bind to `contenteditable` elements.
select.SelectController	The controller for the `<select>` directive. This provides support for reading and writing the selected value(s) of the control and also coordinates dynamically added `<option>` elements, perhaps by an `ngRepeat` directive.
$rootScope.Scope	A root scope can be retrieved using the $rootScope key from the $injector. Child scopes are created using the $new() method. (Most scopes are created automatically when compiled HTML template is executed.)

Provider

Name	Description
$anchorScrollProvider	Use `$anchorScrollProvider` to disable automatic scrolling whenever $location.hash() changes.
$animateProvider	Default implementation of $animate that doesn't perform any animations, instead just synchronously performs DOM updates and resolves the returned runner promise.
$compileProvider	
$controllerProvider	The $controller service is used by Angular to create new controllers.
$filterProvider	Filters are just functions which transform input to an output. However filters need to be Dependency Injected. To achieve this a filter definition consists of a factory function which is annotated with dependencies and is responsible for creating a filter function.
$httpProvider	Use `$httpProvider` to change the default behavior of the $http service.
$interpolateProvider	Used for configuring the interpolation markup. Defaults to `{{` and `}}`.
$locationProvider	Use the `$locationProvider` to configure how the application deep linking paths are stored.
$logProvider	Use the `$logProvider` to configure how the application logs messages

$parseProvider	`$parseProvider` can be used for configuring the default behavior of the $parse service.
$rootScopeProvider	Provider for the $rootScope service.
$sceDelegateProvider	The `$sceDelegateProvider` provider allows developers to configure the $sceDelegate service. This allows one to get/set the whitelists and blacklists used to ensure that the URLs used for sourcing Angular templates are safe. Refer $sceDelegateProvider.resourceUrlWhitelist and $sceDelegateProvider.resourceUrlBlacklist
$sceProvider	The $sceProvider provider allows developers to configure the $sce service. • enable/disable Strict Contextual Escaping (SCE) in a module • override the default implementation with a custom delegate

Service

Name	Description
$anchorScroll	When called, it scrolls to the element related to the specified `hash` or (if omitted) to the current value of $location.hash(), according to the rules specified in the HTML5 spec.
$animate	The $animate service exposes a series of DOM utility methods that provide support for animation hooks. The default behavior is the application of DOM operations, however, when an animation is detected (and animations are enabled), $animate will do the heavy lifting to ensure that animation runs with the triggered DOM operation.
$cacheFactory	Factory that constructs Cache objects and gives access to them.
$templateCache	The first time a template is used, it is loaded in the template cache for quick retrieval. You can load templates directly into the cache in a `script` tag, or by consuming the `$templateCache` service directly.
$compile	Compiles an HTML string or DOM into a template and produces a template function, which can then be used to link `scope` and the template together.
$controller	`$controller` service is responsible for instantiating controllers.
$document	A jQuery or jqLite wrapper for the browser's `window.document` object.
$exceptionHandler	Any uncaught exception in angular expressions is delegated to this service. The default implementation simply delegates to `$log.error` which logs it into the browser console.
$filter	Filters are used for formatting data displayed to the user.

$httpParamSerializer	Default $http params serializer that converts objects to a part of a request URL according to the following rules: • `{'foo': 'bar'}` results in `foo=bar` • `{'foo': Date.now()}` results in `foo=2015-04-01T09%3A50%3A49.262Z` (`toISOString()` and encoded representation of a Date object) • `{'foo': ['bar', 'baz']}` results in `foo=bar&foo=baz` (repeated key for each array element) • `{'foo': {'bar':'baz'}}` results in `foo=%7B%22bar%22%3A%22baz%22%7D"` (stringified and encoded representation of an object)
$httpParamSerializerJQLike	Alternative $http params serializer that follows jQuery's `param()` method logic.
$http	The `$http` service is a core Angular service that facilitates communication with the remote HTTP servers via the browser's XMLHttpRequest object or via JSONP.
$httpBackend	HTTP backend used by the service that delegates to XMLHttpRequest object or JSONP and deals with browser incompatibilities.
$interpolate	Compiles a string with markup into an interpolation function. This service is used by the HTML $compile service for data binding. See $interpolateProvider for configuring the interpolation markup.
$interval	Angular's wrapper for `window.setInterval`. The `fn` function is executed every `delay` milliseconds.
$locale	$locale service provides localization rules for various Angular components. As of right now the only public api is:
$location	The $location service parses the URL in the browser address bar (based on the window.location) and makes the URL available to your application. Changes to the URL in the address bar are reflected into $location service and changes to $location are reflected into the browser address bar.
$log	Simple service for logging. Default implementation safely writes the message into the browser's console (if present).
$parse	Converts Angular expression into a function.
$q	A service that helps you run functions asynchronously, and use their return values (or exceptions) when they are done processing.
$rootElement	The root element of Angular application. This is either the element where ngApp was declared or the element passed into `angular.bootstrap`. The element represents the root element of application. It is also the location where the application's $injector service gets published, and can be retrieved using `$rootElement.injector()`.

$rootScope	Every application has a single root scope. All other scopes are descendant scopes of the root scope. Scopes provide separation between the model and the view, via a mechanism for watching the model for changes. They also provide an event emission/broadcast and subscription facility. See the developer guide on scopes.
$sceDelegate	`$sceDelegate` is a service that is used by the `$sce` service to provide Strict Contextual Escaping (SCE) services to AngularJS.
$sce	`$sce` is a service that provides Strict Contextual Escaping services to AngularJS.
$templateRequest	The `$templateRequest` service downloads the provided template using `$http` and, upon success, stores the contents inside of `$templateCache`. If the HTTP request fails or the response data of the HTTP request is empty, a `$compile` error will be thrown (the exception can be thwarted by setting the 2nd parameter of the function to true).
$timeout	Angular's wrapper for `window.setTimeout`. The `fn` function is wrapped into a try/catch block and delegates any exceptions to $exceptionHandler service.
$window	A reference to the browser's `window` object. While `window` is globally available in JavaScript, it causes testability problems, because it is a global variable. In angular we always refer to it through the `$window` service, so it may be overridden, removed or mocked for testing.

Input

Name	Description
input[text]	Standard HTML text input with angular data binding, inherited by most of the `input` elements.
input[date]	Input with date validation and transformation. In browsers that do not yet support the HTML5 date input, a text element will be used. In that case, text must be entered in a valid ISO-8601 date format (yyyy-MM-dd), for example: `2009-01-06`. Since many modern browsers do not yet support this input type, it is important to provide cues to users on the expected input format via a placeholder or label.
input[datetime-local]	Input with datetime validation and transformation. In browsers that do not yet support the HTML5 date input, a text element will be used. In that case, the text must be entered in a valid ISO-8601 local datetime format (yyyy-MM-ddTHH:mm:ss), for example: `2010-12-28T14:57:00`.
input[time]	Input with time validation and transformation. In browsers that do not yet support the HTML5 date input, a text element will be used. In that case, the text must be entered in a valid ISO-8601 local time format (HH:mm:ss), for example: `14:57:00`. Model must be a Date object. This binding will always

	output a Date object to the model of January 1, 1970, or local date `new Date(1970, 0, 1, HH, mm, ss)`.
input[week]	Input with week-of-the-year validation and transformation to Date. In browsers that do not yet support the HTML5 week input, a text element will be used. In that case, the text must be entered in a valid ISO-8601 week format (yyyy-W##), for example: `2013-W02`.
input[month]	Input with month validation and transformation. In browsers that do not yet support the HTML5 month input, a text element will be used. In that case, the text must be entered in a valid ISO-8601 month format (yyyy-MM), for example: `2009-01`.
input[number]	Text input with number validation and transformation. Sets the `number` validation error if not a valid number.
input[url]	Text input with URL validation. Sets the `url` validation error key if the content is not a valid URL.
input[email]	Text input with email validation. Sets the `email` validation error key if not a valid email address.
input[radio]	HTML radio button.
input[checkbox]	HTML checkbox.

Filter

Name	Description
filter	Selects a subset of items from `array` and returns it as a new array.
currency	Formats a number as a currency (ie $1,234.56). When no currency symbol is provided, default symbol for current locale is used.
number	Formats a number as text.
date	Formats `date` to a string based on the requested `format`.
json	Allows you to convert a JavaScript object into JSON string.
lowercase	Converts string to lowercase.
uppercase	Converts string to uppercase.
limitTo	Creates a new array or string containing only a specified number of elements. The elements are taken from either the beginning or the end of the source array, string or number, as specified by the value and sign (positive or negative) of `limit`. If a number is used as input, it is converted to a string.
orderBy	Orders a specified `array` by the `expression` predicate. It is ordered alphabetically for strings and numerically for numbers. Note: if you notice numbers are not being sorted correctly, make sure they are actually being saved as numbers and not strings.

`ngRoute`

ngRoute

The `ngRoute` module provides routing and deeplinking services and directives for angular apps.

Example

See $route for an example of configuring and using `ngRoute`.

Installation

First include `angular-route.js` in your HTML:

```
<script src="angular.js"> <script src="angular-route.js">
```

You can download this file from the following places:

- Google CDN
  e.g. `//ajax.googleapis.com/ajax/libs/angularjs/X.Y.Z/angular-route.js`
- Bower
  e.g.

```
bower install angular-route@X.Y.Z
```

- code.angularjs.org
  e.g.

```
"//code.angularjs.org/X.Y.Z/angular-route.js"
```

where X.Y.Z is the AngularJS version you are running.

Then load the module in your application by adding it as a dependent module:

```
angular.module('app', ['ngRoute']);
```

With that you're ready to get started!

Module Components

Directive

Name	Description
ngView	# Overview `ngView` is a directive that complements the $route service by including the rendered template of the current route into the main layout (`index.html`) file. Every time the current route changes, the included view changes with it according to the configuration of the `$route` service.

Provider

Name	Description
$routeProvider	Used for configuring routes.

Service

Name	Description
$route	`$route` is used for deep-linking URLs to controllers and views (HTML partials). It watches `$location.url()` and tries to map the path to an existing route definition.
$routeParams	The `$routeParams` service allows you to retrieve the current set of route parameters.

`ngAnimate`

The `ngAnimate` module provides support for CSS-based animations (keyframes and transitions) as well as JavaScript-based animations via callback hooks. Animations are not enabled by default, however, by including `ngAnimate` then the animation hooks are enabled for an Angular app.

Usage

Simply put, there are two ways to make use of animations when ngAnimate is used: by using **CSS** and **JavaScript**. The former works purely based using CSS (by using matching CSS selectors/styles) and the latter triggers animations that are registered via `module.animation()`. For both CSS and JS animations the sole requirement is to have a matching `CSS class` that exists both in the registered animation and within the HTML element that the animation will be triggered on.

Directive Support

The following directives are "animation aware":

Directive	Supported Animations
ngRepeat	enter, leave and move
ngView	enter and leave
ngInclude	enter and leave
ngSwitch	enter and leave
ngIf	enter and leave
ngClass	add and remove (the CSS class(es) present)
ngShow & ngHide	add and remove (the ng-hide class value)
form & ngModel	add and remove (dirty, pristine, valid, invalid & all other validations)
ngMessages	add and remove (ng-active & ng-inactive)
ngMessage	enter and leave

(More information can be found by visiting each the documentation associated with each directive.)

CSS-based Animations

CSS-based animations with ngAnimate are unique since they require no JavaScript code at all. By using a CSS class that we reference between our HTML and CSS code we can create an animation that will be picked up by Angular when an the underlying directive performs an operation.

The example below shows how an `enter` animation can be made possible on a element using `ng-if`:

```
<div ng-if="bool" class="fade"> Fade me in out </div> <button ng-click="bool=true">Fade
In!</button> <button ng-click="bool=false">Fade Out!</button>
```

Notice the CSS class **fade**? We can now create the CSS transition code that references this class:

```
/* The starting CSS styles for the enter animation */ .fade.ng-enter { transition:0.5s linear
all; opacity:0; } /* The starting CSS styles for the enter animation */ .fade.ng-enter.ng-
enter-active { opacity:1; }
```

The key thing to remember here is that, depending on the animation event (which each of the directives above trigger depending on what's going on) two generated CSS classes will be applied to the element; in the example above we have `.ng-enter` and `.ng-enter-active`. For CSS transitions, the transition code **must** be defined within the starting CSS class (in this case `.ng-enter`). The destination class is what the transition will animate towards.

If for example we wanted to create animations for `leave` and `move` (ngRepeat triggers move) then we can do so using the same CSS naming conventions:

```
/* now the element will fade out before it is removed from the DOM */ .fade.ng-leave {
transition:0.5s linear all; opacity:1; } .fade.ng-leave.ng-leave-active { opacity:0; }
```

We can also make use of **CSS Keyframes** by referencing the keyframe animation within the starting CSS class:

```
/* there is no need to define anything inside of the destination CSS class since the keyframe
will take charge of the animation */ .fade.ng-leave { animation: my_fade_animation 0.5s
linear; -webkit-animation: my_fade_animation 0.5s linear; } @keyframes my_fade_animation {
from { opacity:1; } to { opacity:0; } } @-webkit-keyframes my_fade_animation { from {
opacity:1; } to { opacity:0; } }
```

Feel free also mix transitions and keyframes together as well as any other CSS classes on the same element.

CSS Class-based Animations

Class-based animations (animations that are triggered via `ngClass`, `ngShow`, `ngHide` and some other directives) have a slightly different naming convention. Class-based animations are basic enough that a standard transition or keyframe can be referenced on the class being added and removed.

For example if we wanted to do a CSS animation for `ngHide` then we place an animation on the `.ng-hide` CSS class:

```
<div ng-show="bool" class="fade"> Show and hide me </div> <button ng-
click="bool=true">Toggle</button> <style> .fade.ng-hide { transition:0.5s linear all;
opacity:0; } </style>
```

All that is going on here with ngShow/ngHide behind the scenes is the `.ng-hide` class is added/removed (when the hidden state is valid). Since ngShow and ngHide are animation aware then we can match up a transition and ngAnimate handles the rest.

In addition the addition and removal of the CSS class, ngAnimate also provides two helper methods that we can use to further decorate the animation with CSS styles.

```
<div ng-class="{on:onOff}" class="highlight"> Highlight this box </div> <button ng-
click="onOff=!onOff">Toggle</button> <style> .highlight { transition:0.5s linear all; }
.highlight.on-add { background:white; } .highlight.on { background:yellow; } .highlight.on-
remove { background:black; } </style>
```

We can also make use of CSS keyframes by placing them within the CSS classes.

CSS Staggering Animations

A Staggering animation is a collection of animations that are issued with a slight delay in between each successive operation resulting in a curtain-like effect. The ngAnimate module (versions >=1.2) supports staggering animations and the stagger effect can be performed by creating a **ng-EVENT-stagger** CSS class and attaching that class to the base CSS class used for the animation. The style property expected within the stagger class can either be a **transition-delay** or an **animation-delay** property (or both if your animation contains both transitions and keyframe animations).

```
.my-animation.ng-enter { /* standard transition code */ transition: 1s linear all; opacity:0;
} .my-animation.ng-enter-stagger { /* this will have a 100ms delay between each successive
```

```
leave animation */ transition-delay: 0.1s; /* in case the stagger doesn't work then the
duration value must be set to 0 to avoid an accidental CSS inheritance */ transition-duration:
0s; } .my-animation.ng-enter.ng-enter-active { /* standard transition styles */ opacity:1; }
```

Staggering animations work by default in ngRepeat (so long as the CSS class is defined). Outside of ngRepeat, to use staggering animations on your own, they can be triggered by firing multiple calls to the same event on $animate. However, the restrictions surrounding this are that each of the elements must have the same CSS className value as well as the same parent element. A stagger operation will also be reset if one or more animation frames have passed since the multiple calls to `$animate` were fired.

The following code will issue the **ng-leave-stagger** event on the element provided:

```
var kids = parent.children(); $animate.leave(kids[0]); //stagger index=0
$animate.leave(kids[1]); //stagger index=1 $animate.leave(kids[2]); //stagger index=2
$animate.leave(kids[3]); //stagger index=3 $animate.leave(kids[4]); //stagger index=4
window.requestAnimationFrame(function() { //stagger has reset itself $animate.leave(kids[5]);
//stagger index=0 $animate.leave(kids[6]); //stagger index=1 $scope.$digest(); });
```

Stagger animations are currently only supported within CSS-defined animations.

JavaScript-based Animations

ngAnimate also allows for animations to be consumed by JavaScript code. The approach is similar to CSS-based animations (where there is a shared CSS class that is referenced in our HTML code) but in addition we need to register the JavaScript animation on the module. By making use of the `module.animation()` module function we can register the ainmation.

Let's see an example of a enter/leave animation using `ngRepeat`:

```
<div ng-repeat="item in items" class="slide"> {{ item }} </div>
```

See the **slide** CSS class? Let's use that class to define an animation that we'll structure in our module code by using `module.animation`:

```
myModule.animation('.slide', [function() { return { // make note that other events (like
addClass/removeClass) // have different function input parameters enter: function(element,
doneFn) { jQuery(element).fadeIn(1000, doneFn); // remember to call doneFn so that angular //
knows that the animation has concluded }, move: function(element, doneFn) {
jQuery(element).fadeIn(1000, doneFn); }, leave: function(element, doneFn) {
jQuery(element).fadeOut(1000, doneFn); } } }]
```

The nice thing about JS-based animations is that we can inject other services and make use of advanced animation libraries such as greensock.js and velocity.js.

If our animation code class-based (meaning that something like `ngClass`, `ngHide` and `ngShow` triggers it) then we can still define our animations inside of the same registered animation, however, the function input arguments are a bit different:

```
<div ng-class="color" class="colorful"> this box is moody </div> <button ng-
click="color='red'">Change to red</button> <button ng-click="color='blue'">Change to
blue</button> <button ng-click="color='green'">Change to green</button>
```

```
myModule.animation('.colorful', [function() { return { addClass: function(element, className, doneFn) { // do some cool animation and call the doneFn }, removeClass: function(element, className, doneFn) { // do some cool animation and call the doneFn }, setClass: function(element, addedClass, removedClass, doneFn) { // do some cool animation and call the doneFn } } }]
```

CSS + JS Animations Together

AngularJS 1.4 and higher has taken steps to make the amalgamation of CSS and JS animations more flexible. However, unlike earlier versions of Angular, defining CSS and JS animations to work off of the same CSS class will not work anymore. Therefore example below will only result in **JS animations taking charge of the animation**:

```
<div ng-if="bool" class="slide"> Slide in and out </div>

myModule.animation('.slide', [function() { return { enter: function(element, doneFn) { jQuery(element).slideIn(1000, doneFn); } } }]

.slide.ng-enter { transition:0.5s linear all; transform:translateY(-100px); } .slide.ng-enter.ng-enter-active { transform:translateY(0); }
```

Does this mean that CSS and JS animations cannot be used together? Do JS-based animations always have higher priority? We can suppliment for the lack of CSS animations by making use of the `$animateCss` service to trigger our own tweaked-out, CSS-based animations directly from our own JS-based animation code:

```
myModule.animation('.slide', ['$animateCss', function($animateCss) { return { enter: function(element, doneFn) { var animation = $animateCss(element, { event: 'enter' }); if (animation) { // this will trigger `.slide.ng-enter` and `.slide.ng-enter-active`. var runner = animation.start(); runner.done(doneFn); } else { //no CSS animation was detected doneFn(); } } } }]
```

The nice thing here is that we can save bandwidth by sticking to our CSS-based animation code and we don't need to rely on a 3rd-party animation framework.

The `$animateCss` service is very powerful since we can feed in all kinds of extra properties that will be evaluated and fed into a CSS transition or keyframe animation. For example if we wanted to animate the height of an element while adding and removing classes then we can do so by providing that data into `$animateCss` directly:

```
myModule.animation('.slide', ['$animateCss', function($animateCss) { return { enter: function(element, doneFn) { var animation = $animateCss(element, { event: 'enter', addClass: 'maroon-setting', from: { height:0 }, to: { height: 200 } }); if (animation) { animation.start().done(doneFn); } else { doneFn(); } } } }]
```

Now we can fill in the rest via our transition CSS code:

```
/* the transition tells ngAnimate to make the animation happen */ .slide.ng-enter { transition:0.5s linear all; } /* this extra CSS class will be absorbed into the transition since the $animateCss code is adding the class */ .maroon-setting { background:red; }
```

And `$animateCss` will figure out the rest. Just make sure to have the `done()` callback fire the `doneFn` function to signal when the animation is over.

To learn more about what's possible be sure to visit the $animateCss service.

Using $animate in your directive code

So far we've explored how to feed in animations into an Angular application, but how do we trigger animations within our own directives in our application? By injecting the `$animate` service into our directive code, we can trigger structural and class-based hooks which can then be consumed by animations. Let's imagine we have a greeting box that shows and hides itself when the data changes

```
<greeing-box active="onOrOff">Hi there</greeting-box>

ngModule.directive('greetingBox', ['$animate', function($animate) { return function(scope,
element, attrs) { attrs.$observe('active', function(value) { value ?
$animate.addClass(element, 'on') ? $animate.removeClass(element, 'on'); }); }); }]);
```

Now the `on` CSS class is added and removed on the greeting box component. Now if we add a CSS class on top of the greeting box element in our HTML code then we can trigger a CSS or JS animation to happen.

```
/* normally we would create a CSS class to reference on the element */ [greeting-box].on {
transition:0.5s linear all; background:green; color:white; }
```

The `$animate` service contains a variety of other methods like `enter`, `leave`, `animate` and `setClass`. To learn more about what's possible be sure to visit the $animate service API page.

Preventing Collisions With Third Party Libraries

Some third-party frameworks place animation duration defaults across many element or className selectors in order to make their code small and reuseable. This can lead to issues with ngAnimate, which is expecting actual animations on these elements and has to wait for their completion.

You can prevent this unwanted behavior by using a prefix on all your animation classes:

```
/* prefixed with animate- */ .animate-fade-add.animate-fade-add-active { transition:1s linear
all; opacity:0; }
```

You then configure `$animate` to enforce this prefix:

```
$animateProvider.classNameFilter(/animate-/);
```

This also may provide your application with a speed boost since only specific elements containing CSS class prefix will be evaluated for animation when any DOM changes occur in the application.

Callbacks and Promises

When `$animate` is called it returns a promise that can be used to capture when the animation has ended. Therefore if we were to trigger an animation (within our directive code) then we can continue

performing directive and scope related activities after the animation has ended by chaining onto the returned promise that animation method returns.

```
// somewhere within the depths of the directive $animate.enter(element,
parent).then(function() { //the animation has completed });
```

(Note that earlier versions of Angular prior to v1.4 required the promise code to be wrapped using `$scope.$apply(...)`. This is not the case anymore.)

In addition to the animation promise, we can also make use of animation-related callbacks within our directives and controller code by registering an event listener using the `$animate` service. Let's say for example that an animation was triggered on our view routing controller to hook into that:

```
ngModule.controller('HomePageController', ['$animate', function($animate) {
$animate.on('enter', ngViewElement, function(element) { // the animation for this route has
completed }]); }])
```

(Note that you will need to trigger a digest within the callback to get angular to notice any scope-related changes.)

Installation

First include `angular-animate.js` in your HTML:

```
<script src="angular.js"> <script src="angular-animate.js">
```

You can download this file from the following places:

- Google CDN
  e.g. `//ajax.googleapis.com/ajax/libs/angularjs/X.Y.Z/angular-animate.js`
- Bower
  e.g.

  ```
  bower install angular-animate@X.Y.Z
  ```

- code.angularjs.org
  e.g.

  ```
  "//code.angularjs.org/X.Y.Z/angular-animate.js"
  ```

where X.Y.Z is the AngularJS version you are running.

Then load the module in your application by adding it as a dependent module:

```
angular.module('app', ['ngAnimate']);
```

With that you're ready to get started!

Module Components

Service

Name	Description
$animateCss	The `$animateCss` service is a useful utility to trigger customized CSS-based transitions/keyframes from a JavaScript-based animation or directly from a directive. The purpose of `$animateCss` is NOT to side-step how `$animate` and ngAnimate work, but the goal is to allow pre-existing animations or directives to create more complex animations that can be purely driven using CSS code.
$animate	The ngAnimate `$animate` service documentation is the same for the core `$animate` service.

`ngAria`

The `ngAria` module provides support for common ARIA attributes that convey state or semantic information about the application for users of assistive technologies, such as screen readers.

Usage

For ngAria to do its magic, simply include the module as a dependency. The directives supported by ngAria are: `ngModel`, `ngDisabled`, `ngShow`, `ngHide`, `ngClick`, `ngDblClick`, and `ngMessages`.

Below is a more detailed breakdown of the attributes handled by ngAria:

Directive	Supported Attributes
ngDisabled	aria-disabled
ngShow	aria-hidden

ngHide	aria-hidden
ngDblclick	tabindex
ngMessages	aria-live
ngModel	aria-checked, aria-valuemin, aria-valuemax, aria-valuenow, aria-invalid, aria-required, input roles
ngClick	tabindex, keypress event, button role

Find out more information about each directive by reading the ngAria Developer Guide.

Example

Using ngDisabled with ngAria:

```
<md-checkbox ng-disabled="disabled">
```

Becomes:

```
<md-checkbox ng-disabled="disabled" aria-disabled="true">
```

Disabling Attributes

It's possible to disable individual attributes added by ngAria with the config method. For more details, see the Developer Guide.

Installation

First include `angular-aria.js` in your HTML:

```
<script src="angular.js"> <script src="angular-aria.js">
```

You can download this file from the following places:

- Google CDN
  e.g. `//ajax.googleapis.com/ajax/libs/angularjs/X.Y.Z/angular-aria.js`
- Bower
  e.g.

```
bower install angular-aria@X.Y.Z
```

- code.angularjs.org
  e.g.

```
"//code.angularjs.org/X.Y.Z/angular-aria.js"
```

where X.Y.Z is the AngularJS version you are running.

Then load the module in your application by adding it as a dependent module:

```
angular.module('app', ['ngAria']);
```

With that you're ready to get started!

Module Components

Provider

Name	Description
$ariaProvider	Used for configuring the ARIA attributes injected and managed by ngAria.

Service

Name	Description
$aria	

`ngResource`

ngResource

The `ngResource` module provides interaction support with RESTful services via the $resource service.

See `$resource` for usage.

Installation

First include `angular-resource.js` in your HTML:

```
<script src="angular.js"> <script src="angular-resource.js">
```

You can download this file from the following places:

- Google CDN
  e.g. `//ajax.googleapis.com/ajax/libs/angularjs/X.Y.Z/angular-resource.js`
- Bower
  e.g.

```
bower install angular-resource@X.Y.Z
```

- code.angularjs.org
  e.g.

```
"//code.angularjs.org/X.Y.Z/angular-resource.js"
```

where X.Y.Z is the AngularJS version you are running.

Then load the module in your application by adding it as a dependent module:

```
angular.module('app', ['ngResource']);
```

With that you're ready to get started!

Module Components

Service

Name	Description
$resource	A factory which creates a resource object that lets you interact with RESTful server-side data sources.

What is a Module?

You can think of a module as a container for the different parts of your app – controllers, services, filters, directives, etc.

Why?

Most applications have a main method that instantiates and wires together the different parts of the application.

Angular apps don't have a main method. Instead modules declaratively specify how an application should be bootstrapped. There are several advantages to this approach:

- The declarative process is easier to understand.
- You can package code as reusable modules.
- The modules can be loaded in any order (or even in parallel) because modules delay execution.
- Unit tests only have to load relevant modules, which keeps them fast.
- End-to-end tests can use modules to override configuration.

The Basics

I'm in a hurry. How do I get a Hello World module working?

Edit in Plunker
index.html script.js protractor.js

```
<div ng-app="myApp"> <div> {{ 'World' | greet }} </div> </div>

// declare a module var myAppModule = angular.module('myApp', []); // configure the module. //
in this example we will create a greeting filter myAppModule.filter('greet', function() {
return function(name) { return 'Hello, ' + name + '!'; }; });

it('should add Hello to the name', function() { expect(element(by.binding("'World' |
greet")).getText()).toEqual('Hello, World!'); });
```

Important things to notice:

- The Module API
- The reference to `myApp` module in `<div ng-app="myApp">`. This is what bootstraps the app using your module.
- The empty array in `angular.module('myApp', [])`. This array is the list of modules `myApp` depends on.

Recommended Setup

While the example above is simple, it will not scale to large applications. Instead we recommend that you break your application to multiple modules like this:

- A module for each feature
- A module for each reusable component (especially directives and filters)
- And an application level module which depends on the above modules and contains any initialization code.

We've also written a document on how we organize large apps at Google.

The above is a suggestion. Tailor it to your needs.

Edit in Plunker

index.html script.js protractor.js

```
<div ng-controller="XmplController"> {{ greeting }} </div>
```

```
angular.module('xmpl.service', []) .value('greeter', { salutation: 'Hello', localize:
function(localization) { this.salutation = localization.salutation; }, greet: function(name) {
return this.salutation + ' ' + name + '!'; } }) .value('user', { load: function(name) {
this.name = name; } }); angular.module('xmpl.directive', []); angular.module('xmpl.filter',
[]); angular.module('xmpl', ['xmpl.service', 'xmpl.directive', 'xmpl.filter'])
.run(function(greeter, user) { // This is effectively part of the main method initialization
code greeter.localize({ salutation: 'Bonjour' }); user.load('World'); })
.controller('XmplController', function($scope, greeter, user){ $scope.greeting =
greeter.greet(user.name); });
```

```
it('should add Hello to the name', function() {
expect(element(by.binding("greeting")).getText()).toEqual('Bonjour World!'); });
```

Module Loading & Dependencies

A module is a collection of configuration and run blocks which get applied to the application during the bootstrap process. In its simplest form the module consist of a collection of two kinds of blocks:

1. **Configuration blocks** - get executed during the provider registrations and configuration phase. Only providers and constants can be injected into configuration blocks. This is to prevent accidental instantiation of services before they have been fully configured.
2. **Run blocks** - get executed after the injector is created and are used to kickstart the application. Only instances and constants can be injected into run blocks. This is to prevent further system configuration during application run time.

```
angular.module('myModule', []). config(function(injectables) { // provider-injector // This is
an example of config block. // You can have as many of these as you want. // You can only
inject Providers (not instances) // into config blocks. }). run(function(injectables) { //
instance-injector // This is an example of a run block. // You can have as many of these as
you want. // You can only inject instances (not Providers) // into run blocks });
```

Configuration Blocks

There are some convenience methods on the module which are equivalent to the `config` block. For example:

```
angular.module('myModule', []). value('a', 123). factory('a', function() { return 123; }).
directive('directiveName', ...). filter('filterName', ...); // is same as
angular.module('myModule', []). config(function($provide, $compileProvider, $filterProvider) {
$provide.value('a', 123); $provide.factory('a', function() { return 123; });
$compileProvider.directive('directiveName', ...); $filterProvider.register('filterName', ...);
});
```

When bootstrapping, first Angular applies all constant definitions. Then Angular applies configuration blocks in the same order they were registered.

Run Blocks

Run blocks are the closest thing in Angular to the main method. A run block is the code which needs to run to kickstart the application. It is executed after all of the services have been configured and the injector has been created. Run blocks typically contain code which is hard to unit-test, and for this reason should be declared in isolated modules, so that they can be ignored in the unit-tests.

Dependencies

Modules can list other modules as their dependencies. Depending on a module implies that the required module needs to be loaded before the requiring module is loaded. In other words the configuration blocks of the required modules execute before the configuration blocks of the requiring module. The same is true for the run blocks. Each module can only be loaded once, even if multiple other modules require it.

Asynchronous Loading

Modules are a way of managing $injector configuration, and have nothing to do with loading of scripts into a VM. There are existing projects which deal with script loading, which may be used with Angular. Because modules do nothing at load time they can be loaded into the VM in any order and thus script loaders can take advantage of this property and parallelize the loading process.

Creation versus Retrieval

Beware that using `angular.module('myModule', [])` will create the module `myModule` and overwrite any existing module named `myModule`. Use `angular.module('myModule')` to retrieve an existing module.

```
var myModule = angular.module('myModule', []); // add some directives and services
myModule.service('myService', ...); myModule.directive('myDirective', ...); // overwrites both
myService and myDirective by creating a new module var myModule = angular.module('myModule',
[]); // throws an error because myOtherModule has yet to be defined var myModule =
angular.module('myOtherModule');
```

Unit Testing

A unit test is a way of instantiating a subset of an application to apply stimulus to it. Small, structured modules help keep unit tests concise and focused.

Each module can only be loaded once per injector. Usually an Angular app has only one injector and modules are only loaded once. Each test has its own injector and modules are loaded multiple times. In all of these examples we are going to assume this module definition:

```
angular.module('greetMod', []). factory('alert', function($window) { return function(text) {
$window.alert(text); } }). value('salutation', 'Hello'). factory('greet', function(alert,
salutation) { return function(name) { alert(salutation + ' ' + name + '!'); } });
```

Let's write some tests to show how to override configuration in tests.

```
describe('myApp', function() { // load application module (`greetMod`) then load a special //
test module which overrides `$window` with a mock version, // so that calling `window.alert()`
will not block the test // runner with a real alert box. beforeEach(module('greetMod',
function($provide) { $provide.value('$window', { alert: jasmine.createSpy('alert') }); })); //
inject() will create the injector and inject the `greet` and // `$window` into the tests.
it('should alert on $window', inject(function(greet, $window) { greet('World');
expect($window.alert).toHaveBeenCalledWith('Hello World!'); })); // this is another way of
overriding configuration in the // tests using inline `module` and `inject` methods.
it('should alert using the alert service', function() { var alertSpy =
jasmine.createSpy('alert'); module(function($provide) { $provide.value('alert', alertSpy); });
inject(function(greet) { greet('World'); expect(alertSpy).toHaveBeenCalledWith('Hello
World!'); }); }); });
```

What is a Module?

You can think of a module as a container for the different parts of your app – controllers, services, filters, directives, etc.

Why?

Most applications have a main method that instantiates and wires together the different parts of the application.

Angular apps don't have a main method. Instead modules declaratively specify how an application should be bootstrapped. There are several advantages to this approach:

- The declarative process is easier to understand.
- You can package code as reusable modules.
- The modules can be loaded in any order (or even in parallel) because modules delay execution.
- Unit tests only have to load relevant modules, which keeps them fast.
- End-to-end tests can use modules to override configuration.

The Basics

I'm in a hurry. How do I get a Hello World module working?

Edit in Plunker
index.html script.js protractor.js

```
<div ng-app="myApp"> <div> {{ 'World' | greet }} </div> </div>

// declare a module var myAppModule = angular.module('myApp', []); // configure the module. //
in this example we will create a greeting filter myAppModule.filter('greet', function() {
return function(name) { return 'Hello, ' + name + '!'; }; });

it('should add Hello to the name', function() { expect(element(by.binding("'World' |
greet")).getText()).toEqual('Hello, World!'); });
```

Important things to notice:

- The Module API
- The reference to `myApp` module in `<div ng-app="myApp">`. This is what bootstraps the app using your module.
- The empty array in `angular.module('myApp', [])`. This array is the list of modules `myApp` depends on.

Recommended Setup

While the example above is simple, it will not scale to large applications. Instead we recommend that you break your application to multiple modules like this:

- A module for each feature
- A module for each reusable component (especially directives and filters)

- And an application level module which depends on the above modules and contains any initialization code.

We've also written a document on how we organize large apps at Google.

The above is a suggestion. Tailor it to your needs.

Edit in Plunker
index.html script.js protractor.js

```
<div ng-controller="XmplController"> {{ greeting }} </div>

angular.module('xmpl.service', []) .value('greeter', { salutation: 'Hello', localize:
function(localization) { this.salutation = localization.salutation; }, greet: function(name) {
return this.salutation + ' ' + name + '!'; } }) .value('user', { load: function(name) {
this.name = name; } }); angular.module('xmpl.directive', []); angular.module('xmpl.filter',
[]); angular.module('xmpl', ['xmpl.service', 'xmpl.directive', 'xmpl.filter'])
.run(function(greeter, user) { // This is effectively part of the main method initialization
code greeter.localize({ salutation: 'Bonjour' }); user.load('World'); })
.controller('XmplController', function($scope, greeter, user){ $scope.greeting =
greeter.greet(user.name); });

it('should add Hello to the name', function() {
expect(element(by.binding("greeting")).getText()).toEqual('Bonjour World!'); });
```

Module Loading & Dependencies

A module is a collection of configuration and run blocks which get applied to the application during the bootstrap process. In its simplest form the module consist of a collection of two kinds of blocks:

1. **Configuration blocks** - get executed during the provider registrations and configuration phase. Only providers and constants can be injected into configuration blocks. This is to prevent accidental instantiation of services before they have been fully configured.
2. **Run blocks** - get executed after the injector is created and are used to kickstart the application. Only instances and constants can be injected into run blocks. This is to prevent further system configuration during application run time.

```
angular.module('myModule', []). config(function(injectables) { // provider-injector // This is
an example of config block. // You can have as many of these as you want. // You can only
inject Providers (not instances) // into config blocks. }). run(function(injectables) { //
instance-injector // This is an example of a run block. // You can have as many of these as
you want. // You can only inject instances (not Providers) // into run blocks });
```

Configuration Blocks

There are some convenience methods on the module which are equivalent to the `config` block. For example:

```
angular.module('myModule', []). value('a', 123). factory('a', function() { return 123; }).
directive('directiveName', ...). filter('filterName', ...); // is same as
```

```
angular.module('myModule', []). config(function($provide, $compileProvider, $filterProvider) {
$provide.value('a', 123); $provide.factory('a', function() { return 123; });
$compileProvider.directive('directiveName', ...); $filterProvider.register('filterName', ...);
});
```

When bootstrapping, first Angular applies all constant definitions. Then Angular applies configuration blocks in the same order they were registered.

Run Blocks

Run blocks are the closest thing in Angular to the main method. A run block is the code which needs to run to kickstart the application. It is executed after all of the services have been configured and the injector has been created. Run blocks typically contain code which is hard to unit-test, and for this reason should be declared in isolated modules, so that they can be ignored in the unit-tests.

Dependencies

Modules can list other modules as their dependencies. Depending on a module implies that the required module needs to be loaded before the requiring module is loaded. In other words the configuration blocks of the required modules execute before the configuration blocks of the requiring module. The same is true for the run blocks. Each module can only be loaded once, even if multiple other modules require it.

Asynchronous Loading

Modules are a way of managing $injector configuration, and have nothing to do with loading of scripts into a VM. There are existing projects which deal with script loading, which may be used with Angular. Because modules do nothing at load time they can be loaded into the VM in any order and thus script loaders can take advantage of this property and parallelize the loading process.

Creation versus Retrieval

Beware that using `angular.module('myModule', [])` will create the module `myModule` and overwrite any existing module named `myModule`. Use `angular.module('myModule')` to retrieve an existing module.

```
var myModule = angular.module('myModule', []); // add some directives and services
myModule.service('myService', ...); myModule.directive('myDirective', ...); // overwrites both
myService and myDirective by creating a new module var myModule = angular.module('myModule',
[]); // throws an error because myOtherModule has yet to be defined var myModule =
angular.module('myOtherModule');
```

Unit Testing

A unit test is a way of instantiating a subset of an application to apply stimulus to it. Small, structured modules help keep unit tests concise and focused.

Each module can only be loaded once per injector. Usually an Angular app has only one injector and modules are only loaded once. Each test has its own injector and modules are loaded multiple times. In all of these examples we are going to assume this module definition:

```
angular.module('greetMod', []). factory('alert', function($window) { return function(text) {
$window.alert(text); } }). value('salutation', 'Hello'). factory('greet', function(alert,
salutation) { return function(name) { alert(salutation + ' ' + name + '!'); } });
```

Let's write some tests to show how to override configuration in tests.

```
describe('myApp', function() { // load application module (`greetMod`) then load a special //
test module which overrides `$window` with a mock version, // so that calling `window.alert()`
will not block the test // runner with a real alert box. beforeEach(module('greetMod',
function($provide) { $provide.value('$window', { alert: jasmine.createSpy('alert') }); })); //
inject() will create the injector and inject the `greet` and // `$window` into the tests.
it('should alert on $window', inject(function(greet, $window) { greet('World');
expect($window.alert).toHaveBeenCalledWith('Hello World!'); })); // this is another way of
overriding configuration in the // tests using inline `module` and `inject` methods.
it('should alert using the alert service', function() { var alertSpy =
jasmine.createSpy('alert'); module(function($provide) { $provide.value('alert', alertSpy); });
inject(function(greet) { greet('World'); expect(alertSpy).toHaveBeenCalledWith('Hello
World!'); }); }); });
```

ngCookies

ngCookies

The `ngCookies` module provides a convenient wrapper for reading and writing browser cookies.

See `$cookies` and `$cookieStore` for usage.

Installation

First include `angular-cookies.js` in your HTML:

```
<script src="angular.js"> <script src="angular-cookies.js">
```

You can download this file from the following places:

- Google CDN
  e.g. `//ajax.googleapis.com/ajax/libs/angularjs/X.Y.Z/angular-cookies.js`
- Bower
  e.g.

  ```
  bower install angular-cookies@X.Y.Z
  ```

- code.angularjs.org
  e.g.

  ```
  "//code.angularjs.org/X.Y.Z/angular-cookies.js"
  ```

where X.Y.Z is the AngularJS version you are running.

Then load the module in your application by adding it as a dependent module:

```
angular.module('app', ['ngCookies']);
```

With that you're ready to get started!

Module Components

Service

Name	Description
$cookieStore	Provides a key-value (string-object) storage, that is backed by session cookies. Objects put or retrieved from this storage are automatically serialized or deserialized by angular's toJson/fromJson.
$cookies	Provides read/write access to browser's cookies.

Provider

Name	Description
$cookiesProvider	Use `$cookiesProvider` to change the default behavior of the $cookies service.

`ngSanitize`

ngSanitize

The `ngSanitize` module provides functionality to sanitize HTML.

See `$sanitize` for usage.

Installation

First include `angular-sanitize.js` in your HTML:

```
<script src="angular.js"> <script src="angular-sanitize.js">
```

You can download this file from the following places:

- Google CDN
  e.g. `//ajax.googleapis.com/ajax/libs/angularjs/X.Y.Z/angular-sanitize.js`
- Bower
  e.g.

  ```
  bower install angular-sanitize@X.Y.Z
  ```

- code.angularjs.org
  e.g.

  ```
  "//code.angularjs.org/X.Y.Z/angular-sanitize.js"
  ```

where X.Y.Z is the AngularJS version you are running.

Then load the module in your application by adding it as a dependent module:

```
angular.module('app', ['ngSanitize']);
```

With that you're ready to get started!

Module Components

Filter

NameDescription

linky	Finds links in text input and turns them into html links. Supports http/https/ftp/mailto and plain email address links.

Service

Name	Description
$sanitize	The input is sanitized by parsing the HTML into tokens. All safe tokens (from a whitelist) are then serialized back to properly escaped html string. This means that no unsafe input can make it into the returned string, however, since our parser is more strict than a typical browser parser, it's possible that some obscure input, which would be recognized as valid HTML by a browser, won't make it through the sanitizer. The input may also contain SVG markup. The whitelist is configured using the functions `aHrefSanitizationWhitelist` and `imgSrcSanitizationWhitelist` of `$compileProvider`.

ngMock

ngMock

The `ngMock` module provides support to inject and mock Angular services into unit tests. In addition, ngMock also extends various core ng services such that they can be inspected and controlled in a synchronous manner within test code.

Installation

First include `angular-mocks.js` in your HTML:

```
<script src="angular.js"> <script src="angular-mocks.js">
```

You can download this file from the following places:

- Google CDN
  e.g. `//ajax.googleapis.com/ajax/libs/angularjs/X.Y.Z/angular-mocks.js`
- Bower
  e.g.

  ```
  bower install angular-mocks@X.Y.Z
  ```

- code.angularjs.org
  e.g.

  ```
  "//code.angularjs.org/X.Y.Z/angular-mocks.js"
  ```

where X.Y.Z is the AngularJS version you are running.

Then load the module in your application by adding it as a dependent module:

```
angular.module('app', ['ngMock']);
```

With that you're ready to get started!

Module Components

Object

Name	Description
angular.mock	Namespace from 'angular-mocks.js' which contains testing related code.

Provider

Name	Description
$exceptionHandlerProvider	Configures the mock implementation of `$exceptionHandler` to rethrow or to log errors passed to the `$exceptionHandler`.

Service

Name	Description
$exceptionHandler	Mock implementation of `$exceptionHandler` that rethrows or logs errors passed to it. See $exceptionHandlerProvider for configuration information.
$log	Mock implementation of `$log` that gathers all logged messages in arrays (one array per logging level). These arrays are exposed as `logs` property of each of the level-specific log function, e.g. for level `error` the array is exposed as `$log.error.logs`.
$interval	Mock implementation of the $interval service.
$httpBackend	Fake HTTP backend implementation suitable for unit testing applications that use the $http service.
$timeout	This service is just a simple decorator for $timeout service that adds a "flush" and "verifyNoPendingTasks" methods.
$controller	A decorator for `$controller` with additional `bindings` parameter, useful when testing controllers of directives that use `bindToController`.

Type

Name	Description
angular.mock.TzDate	*NOTE*: this is not an injectable instance, just a globally available mock class of `Date`.
$rootScope.Scope	Scope type decorated with helper methods useful for testing. These methods are automatically available on any Scope instance when `ngMock` module is loaded.

Function

Name	Description
angular.mock.dump	*NOTE*: this is not an injectable instance, just a globally available function.
angular.mock.module	*NOTE*: This function is also published on window for easy access. *NOTE*: This function is declared ONLY WHEN running tests with jasmine or mocha
angular.mock.inject	*NOTE*: This function is also published on window for easy access. *NOTE*: This function is declared ONLY WHEN running tests with jasmine or mocha