Developer Guide

Section 1

Conceptual Overview

This section briefly touches on all of the important parts of AngularJS using a simple example. For a more in-depth explanation, see the tutorial.

Concept	Description
Template	HTML with additional markup
Directives	extend HTML with custom attributes and elements
Model	the data shown to the user in the view and with which the user interacts
Scope	context where the model is stored so that controllers, directives and expressions can access it
Expressions	access variables and functions from the scope
Compiler	parses the template and instantiates directives and expressions
Filter	formats the value of an expression for display to the user
View	what the user sees (the DOM)
Data Binding	sync data between the model and the view
Controller	the business logic behind views
Dependency Injection	Creates and wires objects and functions
Injector	dependency injection container
Module	a container for the different parts of an app including controllers, services, filters, directives which configures the Injector
Service	reusable business logic independent of views

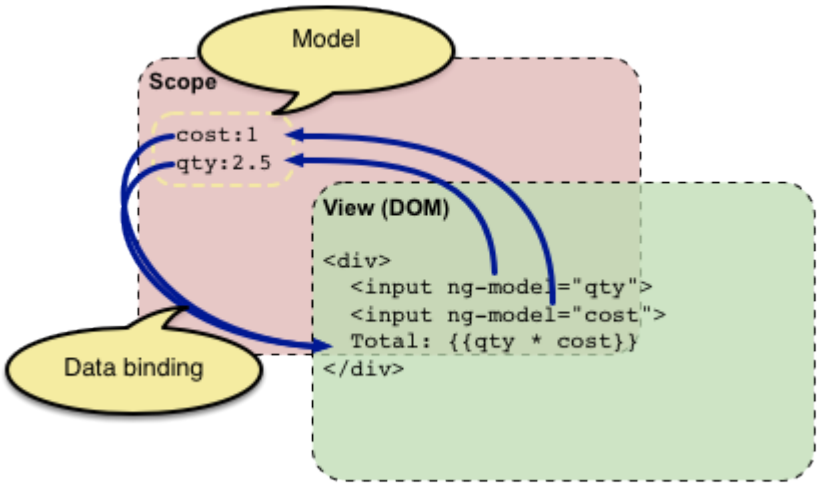
A first example: Data binding

In the following example we will build a form to calculate the costs of an invoice in different currencies.

Let's start with input fields for quantity and cost whose values are multiplied to produce the total of the invoice:

Code: test_1.html

Try out the Live Preview above, and then let's walk through the example and describe what's going on.



This looks like normal HTML, with some new markup. In Angular, a file like this is called a "template". When Angular starts your application, it parses and processes this new markup from the template using the so-called"compiler". The loaded, transformed and rendered DOM is then called the "view".

The first kind of new markup are the so-called "directives". They apply special behavior to attributes or elements in the HTML. In the example above we use the `ng-app` attribute, which is linked to a directive that automatically initializes our application. Angular also defines a directive for the `input`element that adds extra behavior to the element. The `ng-model` directive stores/updates the value of the input field into/from a variable.

Custom directives to access the DOM: In Angular, the only place where an application should access the DOM is within directives. This is important because artifacts that access the DOM are hard to test. If you need to access the DOM directly you should write a custom directive for this. Thedirectives guide explains how to do this.

The second kind of new markup are the double curly braces `{{ expression | filter }}`: When the compiler encounters this markup, it will replace it with the evaluated value of the markup. An "expression" in a template is a JavaScript-like code snippet that allows to read and write variables. Note that those variables are not global variables. Just like variables in a JavaScript function live in a scope, Angular provides a "scope" for the variables accessible to expressions. The values that are stored in variables on the scope are referred to as the "model" in the rest of the documentation. Applied to the example above, the markup directs Angular to "take the data we got from the input widgets and multiply them together".

The example above also contains a "filter". A filter formats the value of an expression for display to the user. In the example above, the filter `currency` formats a number into an output that looks like money.

The important thing in the example is that Angular provides *live* bindings: Whenever the input values change, the value of the expressions are automatically recalculated and the DOM is updated with their values. The concept behind this is "two-way data binding".

Adding UI logic: Controllers

Let's add some more logic to the example that allows us to enter and calculate the costs in different currencies and also pay the invoice.

Code: test_2.html, test_2.js
What changed?

First, there is a new JavaScript file that contains a so-called "controller". More exactly, the file contains a constructor function that creates the actual controller instance. The purpose of controllers is to expose variables and functionality to expressions and directives.

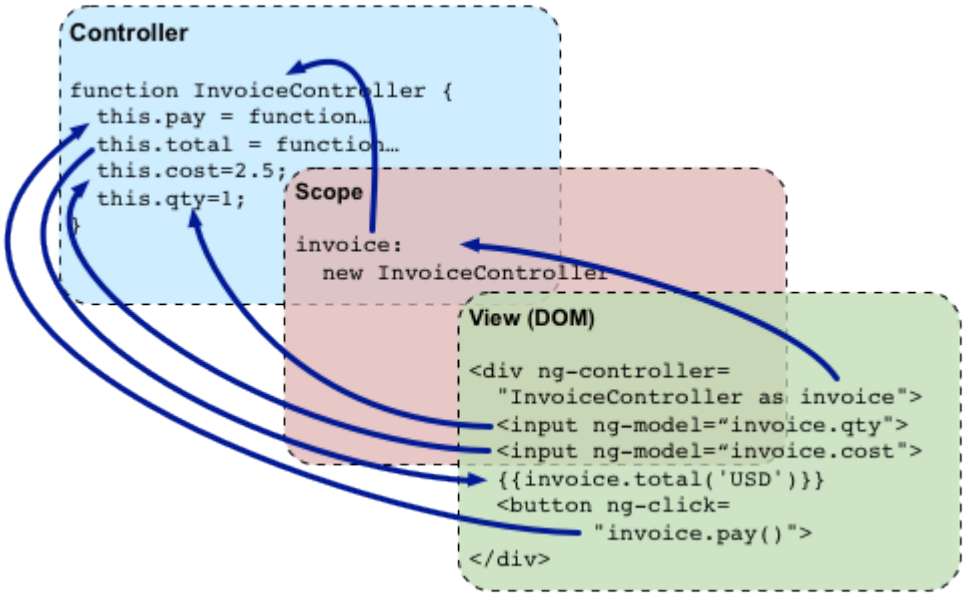
Besides the new file that contains the controller code we also added an `ng-controller` directive to the HTML. This directive tells Angular that the new `InvoiceController` is responsible for the element with the directive and all of the element's children. The syntax `InvoiceController as invoice` tells Angular to instantiate the controller and save it in the variable `invoice` in the current scope.

We also changed all expressions in the page to read and write variables within that controller instance by prefixing them with `invoice.` . The possible currencies are defined in the controller and added to the template using `ng-repeat`. As the controller contains a `total` function we are also able to bind the result of that function to the DOM using `{{ invoice.total(...) }}`.

Again, this binding is live, i.e. the DOM will be automatically updated whenever the result of the function changes. The button to pay the invoice uses the directive `ngClick`. This will evaluate the corresponding expression whenever the button is clicked.

In the new JavaScript file we are also creating a module at which we register the controller. We will talk about modules in the next section.

The following graphic shows how everything works together after we introduced the controller:



View independent business logic: Services

Right now, the `InvoiceController` contains all logic of our example. When the application grows it is a good practice to move view independent logic from the controller into a so called "service", so it can be reused by other parts of the application as well. Later on, we could also change that service to load the exchange rates from the web, e.g. by calling the Yahoo Finance API, without changing the controller.

Let's refactor our example and move the currency conversion into a service in another file:

Code: test_3.html, test_3.js, test_3_finance2.js

What changed? We moved the`convertCurrency` function and the definition of the existing currencies into the new file `finance2.js`. But how does the controller get a hold of the now separated function?

This is where "Dependency Injection" comes into play. Dependency Injection (DI) is a software design pattern that deals with how objects and functions get created and how they get a hold of their dependencies. Everything within Angular (directives, filters, controllers, services, ...) is created and wired using dependency injection. Within Angular, the DI container is called the "injector".

To use DI, there needs to be a place where all the things that should work together are registered. In Angular, this is the purpose of the so-called "modules". When Angular starts, it will use the configuration of the module with the name defined by the `ng-app` directive, including the configuration of all modules that this module depends on.

In the example above: The template contains the directive `ng-app="invoice2"`. This tells Angular to use the `invoice2` module as the main module for the application. The code snippet`angular.module('invoice2', ['finance2'])` specifies that the `invoice2` module depends

on the `finance2` module. By this, Angular uses the `InvoiceController` as well as the `currencyConverter` service.

Now that Angular knows of all the parts of the application, it needs to create them. In the previous section we saw that controllers are created using a factory function. For services there are multiple ways to define their factory (see the service guide). In the example above, we are using a function that returns the `currencyConverter` function as the factory for the service.

Back to the initial question: How does the `InvoiceController` get a reference to the `currencyConverter` function? In Angular, this is done by simply defining arguments on the constructor function. With this, the injector is able to create the objects in the right order and pass the previously created objects into the factories of the objects that depend on them. In our example, the `InvoiceController` has an argument named `currencyConverter`. By this, Angular knows about the dependency between the controller and the service and calls the controller with the service instance as argument.

The last thing that changed in the example between the previous section and this section is that we now pass an array to the `module.controller` function, instead of a plain function. The array first contains the names of the service dependencies that the controller needs. The last entry in the array is the controller constructor function. Angular uses this array syntax to define the dependencies so that the DI also works after minifying the code, which will most probably rename the argument name of the controller constructor function to something shorter like `a`.

Accessing the backend

Let's finish our example by fetching the exchange rates from the Yahoo Finance API. The following example shows how this is done with Angular:

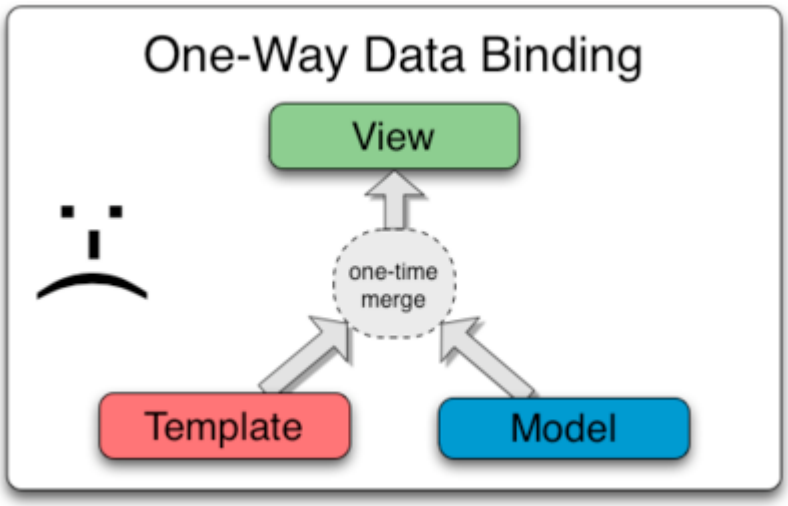
Code: test_4.html, test_4.js, test_4_finance2.js

Section 2

Data Binding

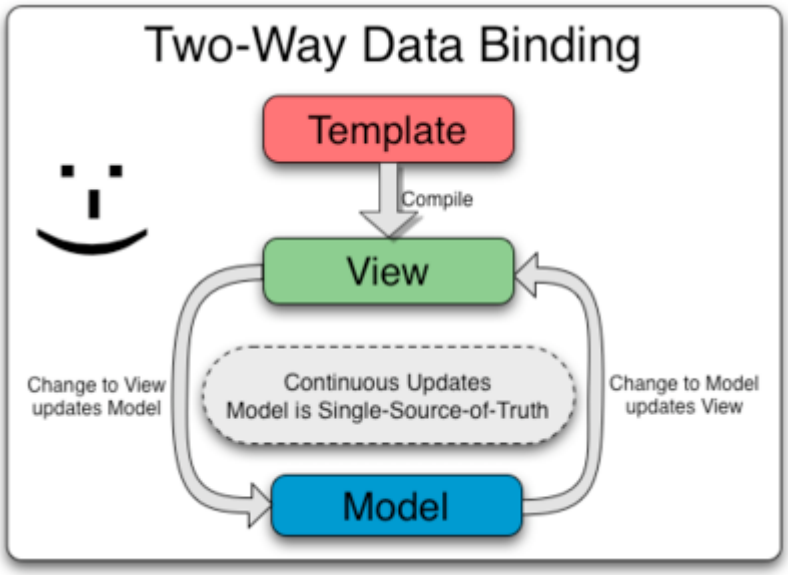
Data-binding in Angular apps is the automatic synchronization of data between the model and view components. The way that Angular implements data-binding lets you treat the model as the single-source-of-truth in your application. The view is a projection of the model at all times. When the model changes, the view reflects the change, and vice versa.

Data Binding in Classical Template Systems



Most templating systems bind data in only one direction: they merge template and model components together into a view. After the merge occurs, changes to the model or related sections of the view are NOT automatically reflected in the view. Worse, any changes that the user makes to the view are not reflected in the model. This means that the developer has to write code that constantly syncs the view with the model and the model with the view.

Data Binding in Angular Templates



Angular templates work differently. First the template (which is the uncompiled HTML along with any additional markup or directives) is compiled on the browser. The compilation step produces a live view. Any changes to the view are immediately reflected in the model, and any changes in the model are propagated to the view. The model is the single-source-of-truth for the application state, greatly

simplifying the programming model for the developer. You can think of the view as simply an instant projection of your model.

Because the view is just a projection of the model, the controller is completely separated from the view and unaware of it. This makes testing a snap because it is easy to test your controller in isolation without the view and the related DOM/browser dependency.

Section 3

Controllers

Understanding Controllers

In Angular, a Controller is a JavaScript **constructor function** that is used to augment the Angular Scope.

When a Controller is attached to the DOM via the ng-controller 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 for most cases anddirectives to encapsulate manual DOM manipulation.
- Format input — Use angular form controls instead.
- Filter output — Use angular filters instead.
- Share code or state across controllers — Use angular 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 `greeting`property 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, `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 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 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).

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 Services sections of this guide.

Associating Controllers with Angular Scope Objects

You can associate Controllers with scope objects implicitly via the ngController directive or $route service.

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 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 `chiliSpicy`method)

- 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 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 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) the`name` property from the previous
- The `GrandChildController` scope, which overrides (hides) both the `timeOfDay` property defined in`MainController` 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.

Section 4

Services

Angular services are substitutable objects that are wired together using dependency injection (DI). You can use services to organize and share code across your app.

Angular services are:

- Lazily instantiated – Angular only instantiates a service when an application component depends on it.
- Singletons – Each component dependent on a service gets a reference to the single instance generated by the service factory.

Angular offers several useful services (like `$http`), but for most applications you'll also want to create your own.

Note: Like other core Angular identifiers, built-in services always start with `$` (e.g. `$http`).

Using a Service

To use an Angular service, you add it as a dependency for the component (controller, service, filter or directive) that depends on the service. Angular's dependency injection subsystem takes care of the rest.

Code: test_8.html, test_8.js

Creating Services

Application developers are free to define their own services by registering the service's name and **service factory function**, with an Angular module.

The **service factory function** generates the single object or function that represents the service to the rest of the application. The object or function returned by the service is injected into any component (controller, service, filter or directive) that specifies a dependency on the service.

Registering Services

Services are registered to modules via the Module API. Typically you use the Module#factory API to register a service:

```
var myModule = angular.module('myModule', []);
myModule.factory('serviceId', function() {
  var shinyNewServiceInstance;
  // factory function body that constructs shinyNewServiceInstance
  return shinyNewServiceInstance;
});
```

Note that you are not registering a **service instance**, but rather a **factory function** that will create this instance when called.

Dependencies

Services can have their own dependencies. Just like declaring dependencies in a controller, you declare dependencies by specifying them in the service's factory function signature.

For more on dependencies, see the dependency injection docs.

The example module below has two services, each with various dependencies:

```
var batchModule = angular.module('batchModule', []);
batchModule.factory('batchLog', ['$interval', '$log', function($interval, $log) {
  var messageQueue = [];
  function log() {
    if (messageQueue.length) {
      $log.log('batchLog messages: ', messageQueue);
      messageQueue = [];
    }
  }
  $interval(log, 50000);
  return function(message) {
    messageQueue.push(message);
  }
}]);
batchModule.factory('routeTemplateMonitor', ['$route', 'batchLog', '$rootScope',
  function($route, batchLog, $rootScope) {
    $rootScope.$on('$routeChangeSuccess', function() {
      batchLog($route.current ? $route.current.template : null);
    });
```

```
}]);
```

In the example, note that:

- The `batchLog` service depends on the built-in `$interval` and `$log` services.
- The `routeTemplateMonitor` service depends on the built-in `$route` service and our custom`batchLog` service.
- Both services use the array notation to declare their dependencies.
- The order of identifiers in the array is the same as the order of argument names in the factory function.

Registering a Service with *`$provide`*

You can also register services via the `$provide` service inside of a module's `config` function:

```
angular.module('myModule', []).config(['$provide', function($provide) {
  $provide.factory('serviceId', function() {
    var shinyNewServiceInstance;
    // factory function body that constructs shinyNewServiceInstance
    return shinyNewServiceInstance;
  });
}]);
```

This technique is often used in unit tests to mock out a service's dependencies.

Section 5

Scopes

What are Scopes?

scope is an object that refers to the application model. It is an execution context for expressions. Scopes are arranged in hierarchical structure which mimic the DOM structure of the application. Scopes can watchexpressions and propagate events.

Scope characteristics

- Scopes provide APIs ($watch) to observe model mutations.

- Scopes provide APIs ($apply) to propagate any model changes through the system into the view from outside of the "Angular realm" (controllers, services, Angular event handlers).
- Scopes can be nested to limit access to the properties of application components while providing access to shared model properties. Nested scopes are either "child scopes" or "isolate scopes". A "child scope" (prototypically) inherits properties from its parent scope. An "isolate scope" does not. See isolated scopesfor more information.
- Scopes provide context against which expressions are evaluated. For example `{{username}}` expression is meaningless, unless it is evaluated against a specific scope which defines the `username` property.

Scope as Data-Model

Scope is the glue between application controller and the view. During the template linking phase thedirectives set up `$watch` expressions on the scope. The `$watch` allows the directives to be notified of property changes, which allows the directive to render the updated value to the DOM.

Both controllers and directives have reference to the scope, but not to each other. This arrangement isolates the controller from the directive as well as from the DOM. This is an important point since it makes the controllers view agnostic, which greatly improves the testing story of the applications.

Code: test_9.html, test_9.js

In the above example notice that the `MyController` assigns `World` to the `username` property of the scope. The scope then notifies the `input` of the assignment, which then renders the input with username pre-filled. This demonstrates how a controller can write data into the scope.

Similarly the controller can assign behavior to scope as seen by the `sayHello` method, which is invoked when the user clicks on the 'greet' button. The `sayHello` method can read the `username` property and create a `greeting` property. This demonstrates that the properties on scope update automatically when they are bound to HTML input widgets.

Logically the rendering of `{{greeting}}` involves:

- retrieval of the scope associated with DOM node where `{{greeting}}` is defined in template. In this example this is the same scope as the scope which was passed into `MyController`. (We will discuss scope hierarchies later.)
- Evaluate the `greeting` expression against the scope retrieved above, and assign the result to the text of the enclosing DOM element.

You can think of the scope and its properties as the data which is used to render the view. The scope is the single source-of-truth for all things view related.

From a testability point of view, the separation of the controller and the view is desirable, because it allows us to test the behavior without being distracted by the rendering details.

```
it('should say hello', function() {
  var scopeMock = {};
  var cntl = new MyController(scopeMock);

  // Assert that username is pre-filled
  expect(scopeMock.username).toEqual('World');

  // Assert that we read new username and greet
  scopeMock.username = 'angular';
  scopeMock.sayHello();
  expect(scopeMock.greeting).toEqual('Hello angular!');
});
```

Scope Hierarchies

Each Angular application has exactly one root scope, but may have several child scopes.

The application can have multiple scopes, because some directives create new child scopes (refer to directive documentation to see which directives create new scopes). When new scopes are created, they are added as children of their parent scope. This creates a tree structure which parallels the DOM where they're attached.

When Angular evaluates `{{name}}`, it first looks at the scope associated with the given element for the`name` property. If no such property is found, it searches the parent scope and so on until the root scope is reached. In JavaScript this behavior is known as prototypical inheritance, and child scopes prototypically inherit from their parents.

This example illustrates scopes in application, and prototypical inheritance of properties. The example is followed by a diagram depicting the scope boundaries.

Code: test_10.html, test_10.js, test_10_style.css
Notice that Angular automatically places `ng-scope` class on elements where scopes are attached. The`<style>` definition in this example highlights in red the new scope locations. The child scopes are necessary because the repeater evaluates `{{name}}` expression, but depending on which scope the expression is evaluated it produces different result. Similarly the evaluation of `{{department}}`prototypically inherits from root scope, as it is the only place where the `department` property is defined.

Retrieving Scopes from the DOM.

Scopes are attached to the DOM as `$scope` data property, and can be retrieved for debugging purposes. (It is unlikely that one would need to retrieve scopes in this way inside the application.) The location where the root scope is attached to the DOM is defined by the location of `ng-app` directive. Typically `ng-app` is placed on the `<html>` element, but it can be placed on other elements as well, if, for example, only a portion of the view needs to be controlled by Angular.

To examine the scope in the debugger:

1. Right click on the element of interest in your browser and select 'inspect element'. You should see the browser debugger with the element you clicked on highlighted.
2. The debugger allows you to access the currently selected element in the console as `$0` variable.
3. To retrieve the associated scope in console execute: `angular.element($0).scope()` or just type $scope

Scope Events Propagation

Scopes can propagate events in similar fashion to DOM events. The event can be broadcasted to the scope children or emitted to scope parents.

Code: test_11.html, test_11.js

Scope Life Cycle

The normal flow of a browser receiving an event is that it executes a corresponding JavaScript callback. Once the callback completes the browser re-renders the DOM and returns to waiting for more events.

When the browser calls into JavaScript the code executes outside the Angular execution context, which means that Angular is unaware of model modifications. To properly process model modifications the execution has to enter the Angular execution context using the `$apply` method. Only model modifications which execute inside the `$apply` method will be properly accounted for by Angular. For example if a directive listens on DOM events, such as `ng-click` it must evaluate the expression inside the `$apply`method.

After evaluating the expression, the `$apply` method performs a `$digest`. In the $digest phase the scope examines all of the `$watch` expressions and compares them with the previous value. This dirty checking is done asynchronously. This means that assignment such as `$scope.username="angular"` will not immediately cause a `$watch` to be notified, instead the `$watch` notification is delayed until the `$digest`phase. This delay is desirable, since it

coalesces multiple model updates into one `$watch` notification as well as it guarantees that during the `$watch` notification no other `$watch`es are running. If a `$watch`changes the value of the model, it will force additional `$digest` cycle.

1. **Creation**

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

2. **Watcher registration**

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

3. **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 $interval services.

4. **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.

5. **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.

Scopes and Directives

During the compilation phase, the compiler matches directives against the DOM template. The directives usually fall into one of two categories:

- Observing directives, such as double-curly expressions `{{expression}}`, register listeners using the $watch() method. This type of directive needs to be notified whenever the expression changes so that it can update the view.
- Listener directives, such as ng-click, register a listener with the DOM. When the DOM listener fires, the directive executes the associated expression and updates the view using the $apply() method.

When an external event (such as a user action, timer or XHR) is received, the associated expression must be applied to the scope through the $apply() method so that all listeners are updated correctly.

Directives that Create Scopes

In most cases, directives and scopes interact but do not create new instances of scope. However, some directives, such as ng-controller and ng-repeat, create new child scopes and attach the child scope to the corresponding DOM element. You can retrieve a scope for any DOM element by using an `angular.element(aDomElement).scope()` method call. See the directives guide for more information about isolate scopes.

Controllers and Scopes

Scopes and controllers interact with each other in the following situations:

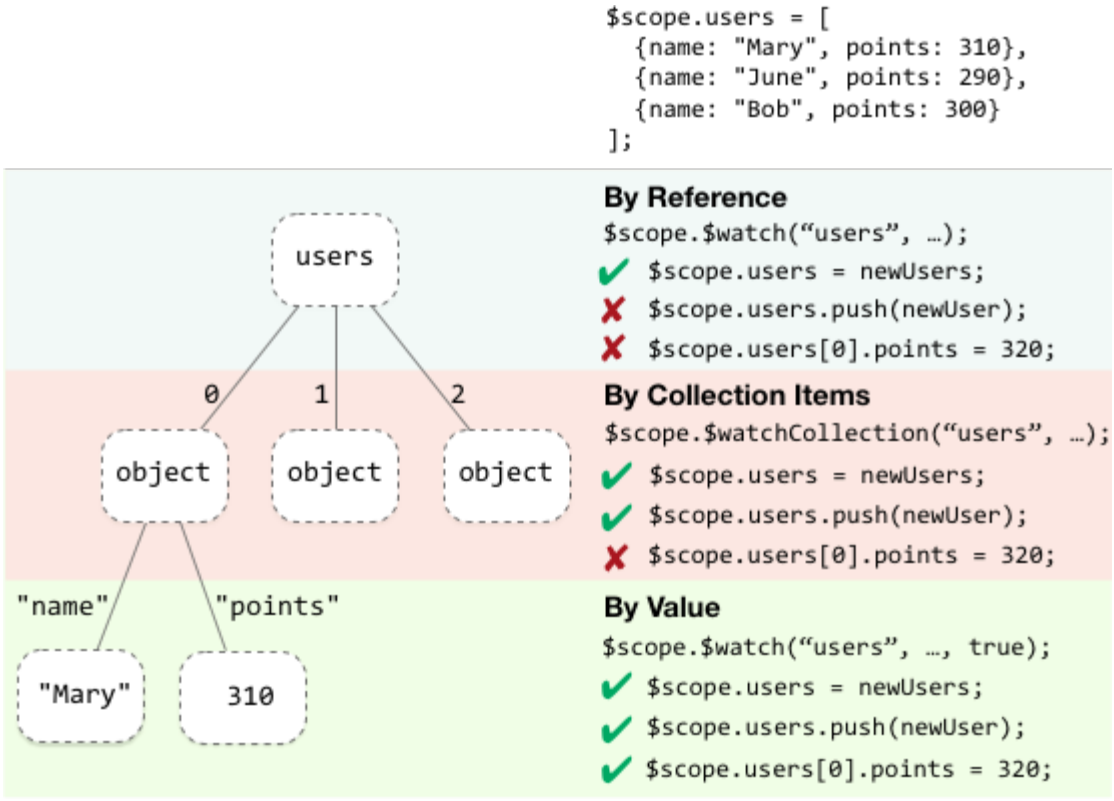
- Controllers use scopes to expose controller methods to templates (see ng-controller).
- Controllers define methods (behavior) that can mutate the model (properties on the scope).
- Controllers may register watches on the model. These watches execute immediately after the controller behavior executes.

See the ng-controller for more information.

Scope *`$watch`* Performance Considerations

Dirty checking the scope for property changes is a common operation in Angular and for this reason the dirty checking function must be efficient. Care should be taken that the dirty checking function does not do any DOM access, as DOM access is orders of magnitude slower than property access on JavaScript object.

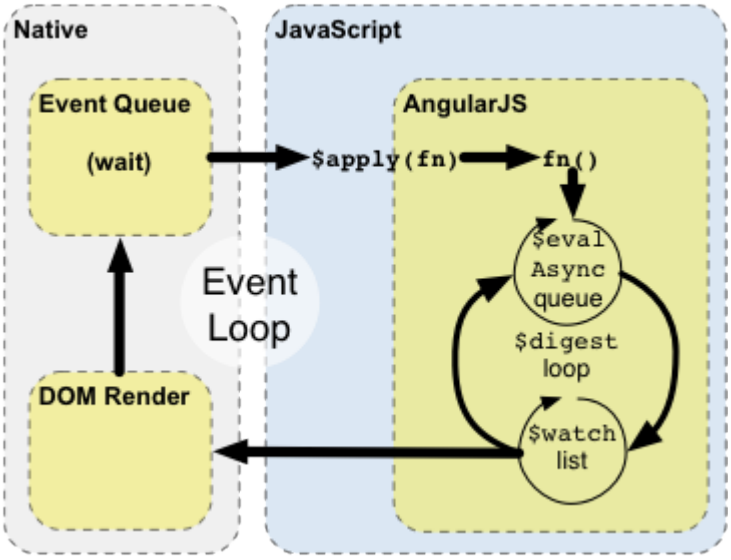
Scope *$watch* Depths



Dirty checking can be done with three strategies: By reference, by collection contents, and by value. The strategies differ in the kinds of changes they detect, and in their performance characteristics.

- Watching *by reference* (scope.$watch `(watchExpression, listener)`) detects a change when the whole value returned by the watch expression switches to a new value. If the value is an array or an object, changes inside it are not detected. This is the most efficient strategy.
- Watching *collection contents* (scope.$watchCollection `(watchExpression, listener)`) detects changes that occur inside an array or an object: When items are added, removed, or reordered. The detection is shallow - it does not reach into nested collections. Watching collection contents is more expensive than watching by reference, because copies of the collection contents need to be maintained. However, the strategy attempts to minimize the amount of copying required.
- Watching *by value* (scope.$watch `(watchExpression, listener, true)`) detects any change in an arbitrarily nested data structure. It is the most powerful change detection strategy, but also the most expensive. A full traversal of the nested data structure is needed on each digest, and a full copy of it needs to be held in memory.

Integration with the browser event loop



The diagram and the example below describe how Angular interacts with the browser's event loop.

1. The browser's event-loop waits for an event to arrive. An event is a user interaction, timer event, or network event (response from a server).
2. The event's callback gets executed. This enters the JavaScript context. The callback can modify the DOM structure.
3. Once the callback executes, the browser leaves the JavaScript context and re-renders the view based on DOM changes.

Angular modifies the normal JavaScript flow by providing its own event processing loop. This splits the JavaScript into classical and Angular execution context. Only operations which are applied in the Angular execution context will benefit from Angular data-binding, exception handling, property watching, etc... You can also use $apply() to enter the Angular execution context from JavaScript. Keep in mind that in most places (controllers, services) $apply has already been called for you by the directive which is handling the event. An explicit call to $apply is needed only when implementing custom event callbacks, or when working with third-party library callbacks.

1. Enter the Angular execution context by calling scope.$apply(`stimulusFn`), where `stimulusFn` is the work you wish to do in the Angular execution context.
2. Angular executes the `stimulusFn()`, which typically modifies application state.
3. Angular enters the $digest loop. The loop is made up of two smaller loops which process$evalAsync queue and the $watch list. The $digest loop keeps iterating until the model stabilizes, which means that the $evalAsync queue is empty and the $watch list does not detect any changes.

4. The $evalAsync queue is used to schedule work which needs to occur outside of current stack frame, but before the browser's view render. This is usually done with `setTimeout(0)`, but the`setTimeout(0)` approach suffers from slowness and may cause view flickering since the browser renders the view after each event.
5. The $watch list is a set of expressions which may have changed since last iteration. If a change is detected then the `$watch` function is called which typically updates the DOM with the new value.
6. Once the Angular $digest loop finishes the execution leaves the Angular and JavaScript context. This is followed by the browser re-rendering the DOM to reflect any changes.

Here is the explanation of how the `Hello world` example achieves the data-binding effect when the user enters text into the text field.

1. During the compilation phase:
    1. the ng-model and input directive set up a `keydown` listener on the `<input>` control.
    2. the interpolation sets up a $watch to be notified of `name` changes.
2. During the runtime phase:
    1. Pressing an 'X' key causes the browser to emit a `keydown` event on the input control.
    2. The input directive captures the change to the input's value and calls$apply`("name = 'X';")` to update the application model inside the Angular execution context.
    3. Angular applies the `name = 'X';` to the model.
    4. The $digest loop begins
    5. The $watch list detects a change on the `name` property and notifies the interpolation, which in turn updates the DOM.
    6. Angular exits the execution context, which in turn exits the `keydown` event and with it the JavaScript execution context.
    7. The browser re-renders the view with update text.

Section 6

Dependency Injection

Dependency Injection (DI) is a software design pattern that deals with how components get hold of their dependencies.

The AngularJS injector subsystem is in charge of creating components, resolving their dependencies, and providing them to other components as requested.

Using Dependency Injection

DI is pervasive throughout AngularJS. You can use it when defining components or when providing `run` and `config` blocks for a module.

- Components such as services, directives, filters, and animations are defined by an injectable factory method or constructor function. These components can be injected with "service" and "value" components as dependencies.
- Controllers are defined by a constructor function, which can be injected with any of the "service" and "value" components as dependencies, but they can also be provided with special dependencies. See Controllers below for a list of these special dependencies.
- The `run` method accepts a function, which can be injected with "service", "value" and "constant" components as dependencies. Note that you cannot inject "providers" into `run` blocks.
- The `config` method accepts a function, which can be injected with "provider" and "constant" components as dependencies. Note that you cannot inject "service" or "value" components into configuration.

See Modules for more details about `run` and `config` blocks.

Factory Methods

The way you define a directive, service, or filter is with a factory function. The factory methods are registered with modules. The recommended way of declaring factories is:

```
angular.module('myModule', [])
.factory('serviceId', ['depService', function(depService) {
  // ...
}])
.directive('directiveName', ['depService', function(depService) {
  // ...
}])
.filter('filterName', ['depService', function(depService) {
  // ...
}]);
```

Module Methods

We can specify functions to run at configuration and run time for a module by calling the `config` and `run` methods. These functions are injectable with dependencies just like the factory functions above.

```
angular.module('myModule', [])
.config(['depProvider', function(depProvider) {
  // ...
}])
.run(['depService', function(depService) {
  // ...
}]);
```

Controllers

Controllers are "classes" or "constructor functions" that are responsible for providing the application behavior that supports the declarative markup in the template. The recommended way of declaring Controllers is using the array notation:

```
someModule.controller('MyController', ['$scope', 'dep1', 'dep2',
function($scope, dep1, dep2) {
  ...
  $scope.aMethod = function() {
    ...
  }
  ...
}]);
```

Unlike services, there can be many instances of the same type of controller in an application.

Moreover, additional dependencies are made available to Controllers:

- `$scope`: Controllers are associated with an element in the DOM and so are provided with access to the scope. Other components (like services) only have access to the `$rootScope` service.
- resolves: If a controller is instantiated as part of a route, then any values that are resolved as part of the route are made available for injection into the controller.

Dependency Annotation

AngularJS invokes certain functions (like service factories and controllers) via the injector. You need to annotate these functions so that the injector knows what services to inject into the function. There are three ways of annotating your code with service name information:

- Using the inline array annotation (preferred)

- Using the `$inject` property annotation
- Implicitly from the function parameter names (has caveats)

Inline Array Annotation

This is the preferred way to annotate application components. This is how the examples in the documentation are written.

For example:

```
someModule.controller('MyController', ['$scope', 'greeter', function($scope,
greeter) {
  // ...
}]);
```

Here we pass an array whose elements consist of a list of strings (the names of the dependencies) followed by the function itself.

When using this type of annotation, take care to keep the annotation array in sync with the parameters in the function declaration.

`$inject` Property Annotation

To allow the minifiers to rename the function parameters and still be able to inject the right services, the function needs to be annotated with the `$inject` property. The `$inject` property is an array of service names to inject.

```
var MyController = function($scope, greeter) {
  // ...
}
MyController.$inject = ['$scope', 'greeter'];
someModule.controller('MyController', MyController);
```

In this scenario the ordering of the values in the `$inject` array must match the ordering of the parameters in `MyController`.

Just like with the array annotation, you'll need to take care to keep the `$inject` in sync with the parameters in the function declaration.

Implicit Annotation

Careful: If you plan to minify your code, your service names will get renamed and break your app.
The simplest way to get hold of the dependencies is to assume that the function parameter names are the names of the dependencies.

```
someModule.controller('MyController', function($scope, greeter) {
  // ...
```

```
});
```

Given a function, the injector can infer the names of the services to inject by examining the function declaration and extracting the parameter names. In the above example, `$scope` and `greeter` are two services which need to be injected into the function.

One advantage of this approach is that there's no array of names to keep in sync with the function parameters. You can also freely reorder dependencies.

However this method will not work with JavaScript minifiers/obfuscators because of how they rename parameters.

Tools like ng-annotate let you use implicit dependency annotations in your app and automatically add inline array annotations prior to minifying. If you decide to take this approach, you probably want to use `ng-strict-di`.

Because of these caveats, we recommend avoiding this style of annotation.

Using Strict Dependency Injection

You can add an `ng-strict-di` directive on the same element as `ng-app` to opt into strict DI mode:

```
<!doctype html>
<html ng-app="myApp" ng-strict-di>
<body>
  I can add: {{ 1 + 2 }}.
  <script src="angular.js"></script>
</body>
</html>
```

Strict mode throws an error whenever a service tries to use implicit annotations.

Consider this module, which includes a `willBreak` service that uses implicit DI:

```
angular.module('myApp', [])
.factory('willBreak', function($rootScope) {
  // $rootScope is implicitly injected
})
.run(['willBreak', function(willBreak) {
  // AngularJS will throw when this runs
}]);
```

When the `willBreak` service is instantiated, AngularJS will throw an error because of strict mode. This is useful when using a tool like ng-annotate to ensure that all of your application components have annotations.

If you're using manual bootstrapping, you can also use strict DI by providing `strictDi: true` in the optional config argument:

```
angular.bootstrap(document, ['myApp'], {
  strictDi: true
});
```

Why Dependency Injection?

This section motivates and explains AngularJS's use of DI. For how to use DI, see above.

For in-depth discussion about DI, see Dependency Injection at Wikipedia, Inversion of Control by Martin Fowler, or read about DI in your favorite software design pattern book.

There are only three ways a component (object or function) can get a hold of its dependencies:

1. The component can create the dependency, typically using the `new` operator.
2. The component can look up the dependency, by referring to a global variable.
3. The component can have the dependency passed to it where it is needed.

The first two options of creating or looking up dependencies are not optimal because they hard code the dependency to the component. This makes it difficult, if not impossible, to modify the dependencies. This is especially problematic in tests, where it is often desirable to provide mock dependencies for test isolation.

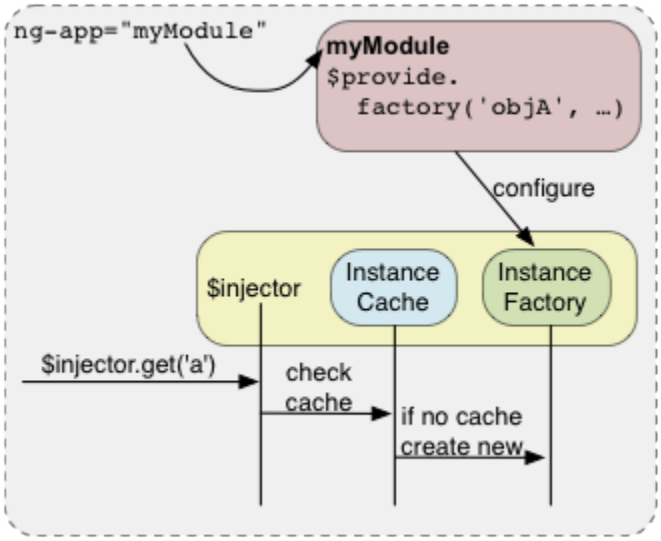
The third option is the most viable, since it removes the responsibility of locating the dependency from the component. The dependency is simply handed to the component.

```
function SomeClass(greeter) {
  this.greeter = greeter;
}


SomeClass.prototype.doSomething = function(name) {
  this.greeter.greet(name);
}
```

In the above example `SomeClass` is not concerned with creating or locating the `greeter` dependency, it is simply handed the `greeter`when it is instantiated.

This is desirable, but it puts the responsibility of getting hold of the dependency on the code that constructs `SomeClass`.



To manage the responsibility of dependency creation, each AngularJS application has an injector. The injector is a service locator that is responsible for construction and lookup of dependencies.

Here is an example of using the injector service:

```
// Provide the wiring information in a module
var myModule = angular.module('myModule', []);
```

Teach the injector how to build a `greeter` service. Notice that `greeter` is dependent on the `$window` service. The `greeter` service is an object that contains a `greet` method.

```
myModule.factory('greeter', function($window) {
  return {
    greet: function(text) {
      $window.alert(text);
    }
  };
});
```

Create a new injector that can provide components defined in our `myModule` module and request our `greeter` service from the injector. (This is usually done automatically by AngularJS bootstrap).

```
var injector = angular.injector(['ng', 'myModule']);
var greeter = injector.get('greeter');
```

Asking for dependencies solves the issue of hard coding, but it also means that the injector needs to be passed throughout the application. Passing the injector breaks the Law of Demeter. To

remedy this, we use a declarative notation in our HTML templates, to hand the responsibility of creating components over to the injector, as in this example:

```
<div ng-controller="MyController">
  <button ng-click="sayHello()">Hello</button>
</div>
function MyController($scope, greeter) {
  $scope.sayHello = function() {
    greeter.greet('Hello World');
  };
}
```

When AngularJS compiles the HTML, it processes the `ng-controller` directive, which in turn asks the injector to create an instance of the controller and its dependencies.

```
injector.instantiate(MyController);
```

This is all done behind the scenes. Notice that by having the `ng-controller` ask the injector to instantiate the class, it can satisfy all of the dependencies of `MyController` without the controller ever knowing about the injector.

This is the best outcome. The application code simply declares the dependencies it needs, without having to deal with the injector. This setup does not break the Law of Demeter.

Section 7

Templates

In AngularJS, templates are written with HTML that contains AngularJS-specific elements and attributes. AngularJS combines the template with information from the model and controller to render the dynamic view that a user sees in the browser.

These are the types of AngularJS elements and attributes you can use:

- Directive — An attribute or element that augments an existing DOM element or represents a reusable DOM component.
- Markup — The double curly brace notation `{{ }}` to bind expressions to elements is built-in AngularJS markup.
- Filter — Formats data for display.
- Form controls — Validates user input.

The following code snippet shows a template with directives and curly-brace expression bindings:

```
<html ng-app>
```

```
 <!-- Body tag augmented with ngController directive  -->
 <body ng-controller="MyController">
   <input ng-model="foo" value="bar">
   <!-- Button tag with ngClick directive, and
          string expression 'buttonText'
          wrapped in "{{ }}" markup -->
   <button ng-click="changeFoo()">{{buttonText}}</button>
   <script src="angular.js"></script>
 </body>
</html>
```

In a simple app, the template consists of HTML, CSS, and AngularJS directives contained in just one HTML file (usually `index.html`).

In a more complex app, you can display multiple views within one main page using "partials" – segments of template located in separate HTML files. You can use the ngView directive to load partials based on configuration passed to the $route service. The AngularJS tutorial shows this technique in steps seven and eight.

Section 8

Expressions

AngularJS expressions are JavaScript-like code snippets that are mainly placed in interpolation bindings such as `<span title="{{ attrBinding }}">{{ textBinding }}</span>`, but also used directly in directive attributes such as `ng-click="functionExpression()"`.

For example, these are valid expressions in AngularJS:

- `1+2`
- `a+b`
- `user.name`
- `items[index]`

AngularJS Expressions vs. JavaScript Expressions

AngularJS expressions are like JavaScript expressions with the following differences:

- **Context:** JavaScript expressions are evaluated against the global `window`. In AngularJS, expressions are evaluated against a `scope` object.

- **Forgiving:** In JavaScript, trying to evaluate undefined properties generates `ReferenceError` or `TypeError`. In AngularJS, expression evaluation is forgiving to `undefined` and `null`.
- **Filters:** You can use filters within expressions to format data before displaying it.
- **No Control Flow Statements:** You cannot use the following in an AngularJS expression: conditionals, loops, or exceptions.
- **No Function Declarations:** You cannot declare functions in an AngularJS expression, even inside `ng-init` directive.
- **No RegExp Creation With Literal Notation:** You cannot create regular expressions in an AngularJS expression. An exception to this rule is `ng-pattern` which accepts valid RegExp.
- **No Object Creation With New Operator:** You cannot use `new` operator in an AngularJS expression.
- **No Bitwise, Comma, And Void Operators:** You cannot use Bitwise, `,` or `void` operators in an AngularJS expression.

If you want to run more complex JavaScript code, you should make it a controller method and call the method from your view. If you want to `eval()` an AngularJS expression yourself, use the `$eval()`method.

Example

Code: test_12.html

You can try evaluating different expressions here:

Code: test_13.html, test_13.js

Context

AngularJS does not use JavaScript's `eval()` to evaluate expressions. Instead AngularJS's $parse service processes these expressions.

AngularJS expressions do not have direct access to global variables like `window`, `document` or `location`. This restriction is intentional. It prevents accidental access to the global state – a common source of subtle bugs.

Instead use services like `$window` and `$location` in functions on controllers, which are then called from expressions. Such services provide mockable access to globals.

It is possible to access the context object using the identifier `this` and the locals object using the identifier `$locals`.

Code: test_14.html, test_14.js

Forgiving

Expression evaluation is forgiving to undefined and null. In JavaScript, evaluating `a.b.c` throws an exception if `a` is not an object. While this makes sense for a general purpose language, the expression evaluations are primarily used for data binding, which often look like this:

```
{{a.b.c}}
```

It makes more sense to show nothing than to throw an exception if `a` is undefined (perhaps we are waiting for the server response, and it will become defined soon). If expression evaluation wasn't forgiving we'd have to write bindings that clutter the code, for example: `{{((a||{}).b||{}).c}}`

Similarly, invoking a function `a.b.c()` on `undefined` or `null` simply returns `undefined`.

No Control Flow Statements

Apart from the ternary operator (`a ? b : c`), you cannot write a control flow statement in an expression. The reason behind this is core to the AngularJS philosophy that application logic should be in controllers, not the views. If you need a real conditional, loop, or to throw from a view expression, delegate to a JavaScript method instead.

No function declarations or RegExp creation with literal notation

You can't declare functions or create regular expressions from within AngularJS expressions. This is to avoid complex model transformation logic inside templates. Such logic is better placed in a controller or in a dedicated filter where it can be tested properly.

`$event`

Directives like `ngClick` and `ngFocus` expose a `$event` object within the scope of that expression. The object is an instance of a jQuery Event Object when jQuery is present or a similar jqLite object.

Code: test_15.html, test_15.js

Note in the example above how we can pass in `$event` to `clickMe`, but how it does not show up in `{{$event}}`. This is because `$event` is outside the scope of that binding.

One-time binding

An expression that starts with `::` is considered a one-time expression. One-time expressions will stop recalculating once they are stable, which happens after the first digest if the expression result is a non-undefined value (see value stabilization algorithm below).

Code: test_16.html, test_16.js

Reasons for using one-time binding

The main purpose of one-time binding expression is to provide a way to create a binding that gets deregistered and frees up resources once the binding is stabilized. Reducing the number of expressions being watched makes the digest loop faster and allows more information to be displayed at the same time.

Value stabilization algorithm

One-time binding expressions will retain the value of the expression at the end of the digest cycle as long as that value is not undefined. If the value of the expression is set within the digest loop and later, within the same digest loop, it is set to undefined, then the expression is not fulfilled and will remain watched.

1. Given an expression that starts with `::`, when a digest loop is entered and expression is dirty-checked, store the value as V
2. If V is not undefined, mark the result of the expression as stable and schedule a task to deregister the watch for this expression when we exit the digest loop
3. Process the digest loop as normal
4. When digest loop is done and all the values have settled, process the queue of watch deregistration tasks. For each watch to be deregistered, check if it still evaluates to a value that is not `undefined`. If that's the case, deregister the watch. Otherwise, keep dirty-checking the watch in the future digest loops by following the same algorithm starting from step 1

Special case for object literals

Unlike simple values, object-literals are watched until every key is defined.
See http://www.bennadel.com/blog/2760-one-time-data-bindings-for-object-literal-expressions-in-angularjs-1-3.htm

How to benefit from one-time binding

If the expression will not change once set, it is a candidate for one-time binding. Here are three example cases.

When interpolating text or attributes:

```
<div name="attr: {{::color}}">text: {{::name | uppercase}}</div>
```

When using a directive with bidirectional binding and parameters that will not change:

```
someModule.directive('someDirective', function() {
  return {
    scope: {
      name: '=',
      color: '@'
    },
    template: '{{name}}: {{color}}'
  };
});
<div some-directive name="::myName" color="My color is {{::myColor}}"></div>
```

When using a directive that takes an expression:

```
<ul>
  <li ng-repeat="item in ::items | orderBy:'name'">{{item.name}};</li>
</ul>
```

Section 9

Interpolation

Interpolation markup with embedded expressions is used by AngularJS to provide data-binding to text nodes and attribute values.

An example of interpolation is shown below:

```
<a ng-href="img/{{username}}.jpg">Hello {{username}}!</a>
```

How text and attribute bindings work

During the compilation process the compiler uses the $interpolate service to see if text nodes and element attributes contain interpolation markup with embedded expressions.

If that is the case, the compiler adds an interpolateDirective to the node and registers watches on the computed interpolation function, which will update the corresponding text nodes or attribute values as part of the normal digest cycle.

Note that the interpolateDirective has a priority of 100 and sets up the watch in the preLink function.

How the string representation is computed

If the interpolated value is not a `String`, it is computed as follows:

- `undefined` and `null` are converted to `''`
- if the value is an object that is not a `Number`, `Date` or `Array`, $interpolate looks for a custom `toString()` function on the object, and uses that. Custom means that `myObject.toString !== Object.prototype.toString`.
- if the above doesn't apply, `JSON.stringify` is used.

Binding to boolean attributes

Attributes such as `disabled` are called `boolean` attributes, because their presence means `true` and their absence means `false`. We cannot use normal attribute bindings with them, because the HTML specification does not require browsers to preserve the values of boolean attributes. This means that if we put an AngularJS interpolation expression into such an attribute then the binding information would be lost, because the browser ignores the attribute value.

In the following example, the interpolation information would be ignored and the browser would simply interpret the attribute as present, meaning that the button would always be disabled.

```
Disabled: <input type="checkbox" ng-model="isDisabled" />
<button disabled="{{isDisabled}}">Disabled</button>
```

For this reason, AngularJS provides special `ng`-prefixed directives for the following boolean attributes: `disabled`, `required`, `selected`, `checked`, `readOnly` , and `open`.

These directives take an expression inside the attribute, and set the corresponding boolean attribute to true when the expression evaluates to truthy.

```
Disabled: <input type="checkbox" ng-model="isDisabled" />
<button ng-disabled="isDisabled">Disabled</button>
```

`ngAttr` for binding to arbitrary attributes

Web browsers are sometimes picky about what values they consider valid for attributes.

For example, considering this template:

```
<svg>
  <circle cx="{{cx}}"></circle>
```

```
</svg>
```

We would expect AngularJS to be able to bind to this, but when we check the console we see something like `Error: Invalid value for attribute cx="{{cx}}"`. Because of the SVG DOM API's restrictions, you cannot simply write `cx="{{cx}}"`.

With `ng-attr-cx` you can work around this problem.

If an attribute with a binding is prefixed with the `ngAttr` prefix (denormalized as `ng-attr-`) then during the binding it will be applied to the corresponding unprefixed attribute. This allows you to bind to attributes that would otherwise be eagerly processed by browsers (e.g. an SVG element's `circle[cx]` attributes). When using `ngAttr`, the `allOrNothing` flag of $interpolate is used, so if any expression in the interpolated string results in `undefined`, the attribute is removed and not added to the element.

For example, we could fix the example above by instead writing:

```
<svg>
  <circle ng-attr-cx="{{cx}}"></circle>
</svg>
```

If one wants to modify a camelcased attribute (SVG elements have valid camelcased attributes), such as `viewBox` on the `svg` element, one can use underscores to denote that the attribute to bind to is naturally camelcased.

For example, to bind to `viewBox`, we can write:

```
<svg ng-attr-view_box="{{viewBox}}">
</svg>
```

Other attributes may also not work as expected when they contain interpolation markup, and can be used with `ngAttr` instead. The following is a list of known problematic attributes:

- **size** in `<select>` elements (see issue 1619)
- **placeholder** in `<textarea>` in Internet Explorer 10/11 (see issue 5025)
- **type** in `<button>` in Internet Explorer 11 (see issue 14117)
- **value** in `<progress>` in Internet Explorer = 11 (see issue 7218)

Known Issues

Dynamically changing an interpolated value

You should avoid dynamically changing the content of an interpolated string (e.g. attribute value or text node). Your changes are likely to be overwritten, when the original string gets evaluated. This restriction applies to both directly changing the content via JavaScript or indirectly using a directive.

For example, you should not use interpolation in the value of the `style` attribute (e.g.`style="color: {{ 'orange' }}; font-weight: {{ 'bold' }};"`) **and** at the same time use a directive that changes the content of that attribute, such as `ngStyle`.

Embedding interpolation markup inside expressions

Note: AngularJS directive attributes take either expressions *or* interpolation markup with embedded expressions. It is considered **bad practice** to embed interpolation markup inside an expression:

```
<div ng-show="form{{$index}}.$invalid"></div>
```

You should instead delegate the computation of complex expressions to the scope, like this:

```
<div ng-show="getForm($index).$invalid"></div>
function getForm(index) {
  return $scope['form' + index];
}
```

You can also access the `scope` with `this` in your templates:

```
<div ng-show="this['form' + $index].$invalid"></div>
```

Why mixing interpolation and expressions is bad practice:

- It increases the complexity of the markup
- There is no guarantee that it works for every directive, because interpolation itself is a directive. If another directive accesses attribute data before interpolation has run, it will get the raw interpolation markup and not data.
- It impacts performance, as interpolation adds another watcher to the scope.
- Since this is not recommended usage, we do not test for this, and changes to AngularJS core may break your code.

Section 10

Filters

Filters format the value of an expression for display to the user. They can be used in view templates, controllers or services. AngularJS comes with a collection of built-in filters, but it is easy to define your own as well.

The underlying API is the `$filterProvider`.

Using filters in view templates

Filters can be applied to expressions in view templates using the following syntax:

```
{{ expression | filter }}
```

E.g. the markup `{{ 12 | currency }}` formats the number 12 as a currency using the `currency` filter. The resulting value is `$12.00`.

Filters can be applied to the result of another filter. This is called "chaining" and uses the following syntax:

```
{{ expression | filter1 | filter2 | ... }}
```

Filters may have arguments. The syntax for this is

```
{{ expression | filter:argument1:argument2:... }}
```

E.g. the markup `{{ 1234 | number:2 }}` formats the number 1234 with 2 decimal points using the `number` filter. The resulting value is `1,234.00`.

When filters are executed

In templates, filters are only executed when their inputs have changed. This is more performant than executing a filter on each `$digest` as is the case with expressions.

There are two exceptions to this rule:

1. In general, this applies only to filters that take primitive values as inputs. Filters that receive Objects as input are executed on each `$digest`, as it would be too costly to track if the inputs have changed.
2. Filters that are marked as `$stateful` are also executed on each $digest. See Stateful filters for more information. Note that no AngularJS core filters are $stateful.

Using filters in controllers, services, and directives

You can also use filters in controllers, services, and directives.

For this, inject a dependency with the name `<filterName>Filter` into your controller/service/directive. E.g. a filter called `number` is injected by using the dependency `numberFilter`. The injected argument is a function that takes the value to format as first argument, and filter parameters starting with the second argument.
The example below uses the filter called `filter`. This filter reduces arrays into sub arrays based on conditions. The filter can be applied in the view template with markup like `{{ctrl.array | filter:'a'}}`, which would do a fulltext search for "a". However, using a

filter in a view template will reevaluate the filter on every digest, which can be costly if the array is big.

The example below therefore calls the filter directly in the controller. By this, the controller is able to call the filter only when needed (e.g. when the data is loaded from the backend or the filter expression is changed).

Code: test_17.html, test_17.js

Creating custom filters

Writing your own filter is very easy: just register a new filter factory function with your module. Internally, this uses the `filterProvider`. This factory function should return a new filter function which takes the input value as the first argument. Any filter arguments are passed in as additional arguments to the filter function.

The filter function should be a pure function, which means that it should always return the same result given the same input arguments and should not affect external state, for example, other AngularJS services. AngularJS relies on this contract and will by default execute a filter only when the inputs to the function change. Stateful filters are possible, but less performant.

Note: Filter names must be valid AngularJS `Expressions` identifiers, such as `uppercase` or `orderBy`. Names with special characters, such as hyphens and dots, are not allowed. If you wish to namespace your filters, then you can use capitalization (`myappSubsectionFilterx`) or underscores (`myapp_subsection_filterx`).
The following sample filter reverses a text string. In addition, it conditionally makes the text upper-case.

Code: test_18.html, test_18.js

Stateful filters

It is strongly discouraged to write filters that are stateful, because the execution of those can't be optimized by AngularJS, which often leads to performance issues. Many stateful filters can be converted into stateless filters just by exposing the hidden state as a model and turning it into an argument for the filter.

If you however do need to write a stateful filter, you have to mark the filter as `$stateful`, which means that it will be executed one or more times during the each `$digest` cycle.

Code: test_19.html, test_19.js

Section 11

Forms

Controls (`input`, `select`, `textarea`) are ways for a user to enter data. A Form is a collection of controls for the purpose of grouping related controls together.

Form and controls provide validation services, so that the user can be notified of invalid input before submitting a form. This provides a better user experience than server-side validation alone because the user gets instant feedback on how to correct the error. Keep in mind that while client-side validation plays an important role in providing good user experience, it can easily be circumvented and thus can not be trusted. Server-side validation is still necessary for a secure application.

Simple form

The key directive in understanding two-way data-binding is ngModel. The `ngModel` directive provides the two-way data-binding by synchronizing the model to the view, as well as view to the model. In addition it provides an API for other directives to augment its behavior.

Code: test_20.html, test_20.js

Note that `novalidate` is used to disable browser's native form validation.

The value of `ngModel` won't be set unless it passes validation for the input field. For example: inputs of type `email` must have a value in the form of `user@domain`.

Using CSS classes

To allow styling of form as well as controls, `ngModel` adds these CSS classes:

- `ng-valid`: the model is valid
- `ng-invalid`: the model is invalid
- `ng-valid-[key]`: for each valid key added by `$setValidity`
- `ng-invalid-[key]`: for each invalid key added by `$setValidity`
- `ng-pristine`: the control hasn't been interacted with yet
- `ng-dirty`: the control has been interacted with
- `ng-touched`: the control has been blurred
- `ng-untouched`: the control hasn't been blurred
- `ng-pending`: any `$asyncValidators` are unfulfilled

The following example uses the CSS to display validity of each form control. In the example both `user.name` and `user.email` are required, but are rendered with red background only after the

input is blurred (loses focus). This ensures that the user is not distracted with an error until after interacting with the control, and failing to satisfy its validity.

Code: test_21.html, test_21.js

Binding to form and control state

A form is an instance of FormController. The form instance can optionally be published into the scope using the `name` attribute.

Similarly, an input control that has the ngModel directive holds an instance of NgModelController. Such a control instance can be published as a property of the form instance using the `name` attribute on the input control. The name attribute specifies the name of the property on the form instance.

This implies that the internal state of both the form and the control is available for binding in the view using the standard binding primitives.

This allows us to extend the above example with these features:

- Custom error message displayed after the user interacted with a control (i.e. when `$touched` is set)
- Custom error message displayed upon submitting the form (`$submitted` is set), even if the user didn't interact with a control

Code: test_22.html, test_22.js

Custom model update triggers

By default, any change to the content will trigger a model update and form validation. You can override this behavior using the ngModelOptions directive to bind only to specified list of events. I.e. `ng-model-options="{ updateOn: 'blur' }"` will update and validate only after the control loses focus. You can set several events using a space delimited list. I.e. `ng-model-options="{ updateOn: 'mousedown blur' }"`

If you want to keep the default behavior and just add new events that may trigger the model update and validation, add "default" as one of the specified events.

I.e. `ng-model-options="{ updateOn: 'default blur' }"`

The following example shows how to override immediate updates. Changes on the inputs within the form will update the model only when the control loses focus (blur event).

Code: test_23.html, test_23.js

Non-immediate (debounced) model updates

You can delay the model update/validation by using the `debounce` key with the ngModelOptions directive. This delay will also apply to parsers, validators and model flags like `$dirty` or `$pristine`.

I.e. `ng-model-options="{ debounce: 500 }"` will wait for half a second since the last content change before triggering the model update and form validation.

If custom triggers are used, custom debouncing timeouts can be set for each event using an object in `debounce`. This can be useful to force immediate updates on some specific circumstances (like blur events).

I.e. `ng-model-options="{ updateOn: 'default blur', debounce: { default: 500, blur: 0 } }"`

If those attributes are added to an element, they will be applied to all the child elements and controls that inherit from it unless they are overridden.

This example shows how to debounce model changes. Model will be updated only 250 milliseconds after last change.

Code: test_24.html, test_24.js

Custom Validation

AngularJS provides basic implementation for most common HTML5 input types: (text, number, url, email, date, radio, checkbox), as well as some directives for validation (`required`, `pattern`, `minlength`, `maxlength`, `min`, `max`).

With a custom directive, you can add your own validation functions to the `$validators` object on the `ngModelController`. To get a hold of the controller, you require it in the directive as shown in the example below.

Each function in the `$validators` object receives the `modelValue` and the `viewValue` as parameters. AngularJS will then call `$setValidity` internally with the function's return value (`true`: valid, `false`: invalid). The validation functions are executed every time an input is changed (`$setViewValue` is called) or whenever the bound `model` changes. Validation happens after successfully running `$parsers` and `$formatters`, respectively. Failed validators are stored by key in `ngModelController.$error`.

Additionally, there is the `$asyncValidators` object which handles asynchronous validation, such as making an `$http` request to the backend. Functions added to the object must return a promise that must be `resolved` when valid or `rejected` when invalid. In-progress async validations are stored by key in `ngModelController.$pending`.

In the following example we create two directives:

- An `integer` directive that validates whether the input is a valid integer. For example, `1.23` is an invalid value, since it contains a fraction. Note that we validate the viewValue (the string value of the control), and not the modelValue. This is because input[number] converts the viewValue to a number when running the `$parsers`.
- A `username` directive that asynchronously checks if a user-entered value is already taken. We mock the server request with a `$q`deferred.

Code: test_25.html, test_25.js

Modifying built-in validators

Since AngularJS itself uses `$validators`, you can easily replace or remove built-in validators, should you find it necessary. The following example shows you how to overwrite the email validator in `input[email]` from a custom directive so that it requires a specific top-level domain, `example.com` to be present. Note that you can alternatively use `ng-pattern` to further restrict the validation.

Code: test_26.html, test_26.js

Implementing custom form controls (using `ngModel`)

AngularJS implements all of the basic HTML form controls (input, select, textarea), which should be sufficient for most cases. However, if you need more flexibility, you can write your own form control as a directive.

In order for custom control to work with `ngModel` and to achieve two-way data-binding it needs to:

- implement `$render` method, which is responsible for rendering the data after it passed the `NgModelController.$formatters`,
- call `$setViewValue` method, whenever the user interacts with the control and model needs to be updated. This is usually done inside a DOM Event listener.

See `$compileProvider.directive` for more info.

The following example shows how to add two-way data-binding to contentEditable elements.

Code: test_27.html, test_27.js

Section 12

Directives

This document explains when you'd want to create your own directives in your AngularJS app, and how to implement them.

What are Directives?

At a high level, directives are markers on a DOM element (such as an attribute, element name, comment or CSS class) that tell AngularJS's **HTML compiler** (`$compile`) to attach a specified behavior to that DOM element (e.g. via event listeners), or even to transform the DOM element and its children.

AngularJS comes with a set of these directives built-in, like `ngBind`, `ngModel`, and `ngClass`. Much like you create controllers and services, you can create your own directives for AngularJS to use. When AngularJS bootstraps your application, the HTML compiler traverses the DOM matching directives against the DOM elements.

What does it mean to "compile" an HTML template? For AngularJS, "compilation" means attaching directives to the HTML to make it interactive. The reason we use the term "compile" is that the recursive process of attaching directives mirrors the process of compiling source code incompiled programming languages.

Matching Directives

Before we can write a directive, we need to know how AngularJS's HTML compiler determines when to use a given directive.

Similar to the terminology used when an element **matches** a selector, we say an element **matches** a directive when the directive is part of its declaration.

In the following example, we say that the `<input>` element **matches** the `ngModel` directive

```
<input ng-model="foo">
```

The following `<input>` element also **matches** `ngModel`:

```
<input data-ng-model="foo">
```

And the following `<person>` element **matches** the `person` directive:

```
<person>{{name}}</person>
```

Normalization

AngularJS **normalizes** an element's tag and attribute name to determine which elements match which directives. We typically refer to directives by their case-sensitive camelCase **normalized** name (e.g. `ngModel`). However, since HTML is case-insensitive, we refer to directives in the DOM by lower-case forms, typically using dash-delimited attributes on DOM elements (e.g. `ng-model`).

The **normalization** process is as follows:

1. Strip `x-` and `data-` from the front of the element/attributes.
2. Convert the `:`, `-`, or `_`-delimited name to `camelCase`.

For example, the following forms are all equivalent and match the `ngBind` directive:

Code: test_28.html, test_28.js

Best Practice: Prefer using the dash-delimited format (e.g. `ng-bind` for `ngBind`). If you want to use an HTML validating tool, you can instead use the `data`-prefixed version (e.g. `data-ng-bind` for `ngBind`). The other forms shown above are accepted for legacy reasons but we advise you to avoid them.

Directive types

`$compile` can match directives based on element names (E), attributes (A), class names (C), and comments (M).

The built-in AngularJS directives show in their documentation page which type of matching they support.

The following demonstrates the various ways a directive (`myDir` in this case) that matches all 4 types can be referenced from within a template.

```
<my-dir></my-dir>
<span my-dir="exp"></span>
<!-- directive: my-dir exp -->
<span class="my-dir: exp;"></span>
```

A directive can specify which of the 4 matching types it supports in the `restrict` property of the directive definition object. The default is `EA`.

Best Practice: Prefer using directives via tag name and attributes over comment and class names. Doing so generally makes it easier to determine what directives a given element matches.
Best Practice: Comment directives were commonly used in places where the DOM API limits the ability to create directives that spanned multiple elements (e.g. inside `<table>` elements).

AngularJS 1.2 introduces `ng-repeat-start` and `ng-repeat-end` as a better solution to this problem. Developers are encouraged to use this over custom comment directives when possible.

Creating Directives

First let's talk about the API for registering directives. Much like controllers, directives are registered on modules. To register a directive, you use the `module.directive` API. `module.directive` takes thenormalized directive name followed by a **factory function.** This factory function should return an object with the different options to tell `$compile` how the directive should behave when matched.

The factory function is invoked only once when the compiler matches the directive for the first time. You can perform any initialization work here. The function is invoked using $injector.invoke which makes it injectable just like a controller.

We'll go over a few common examples of directives, then dive deep into the different options and compilation process.

Best Practice: In order to avoid collisions with some future standard, it's best to prefix your own directive names. For instance, if you created a `<carousel>` directive, it would be problematic if HTML7 introduced the same element. A two or three letter prefix (e.g. `btfCarousel`) works well. Similarly, do not prefix your own directives with `ng` or they might conflict with directives included in a future version of AngularJS.
For the following examples, we'll use the prefix `my` (e.g. `myCustomer`).

Template-expanding directive

Let's say you have a chunk of your template that represents a customer's information. This template is repeated many times in your code. When you change it in one place, you have to change it in several others. This is a good opportunity to use a directive to simplify your template.

Let's create a directive that simply replaces its contents with a static template:

Code: test_29.html, test_29.js

Notice that we have bindings in this directive. After `$compile` compiles and links`<div my-customer></div>`, it will try to match directives on the element's children. This means you can compose directives of other directives. We'll see how to do that in an example below.

In the example above we in-lined the value of the `template` option, but this will become annoying as the size of your template grows.

Best Practice: Unless your template is very small, it's typically better to break it apart into its own HTML file and load it with the `templateUrl` option.
If you are familiar with `ngInclude`, `templateUrl` works just like it. Here's the same example using `templateUrl` instead:

Code: test_30.html, test_30.js

`templateUrl` can also be a function which returns the URL of an HTML template to be loaded and used for the directive. AngularJS will call the `templateUrl` function with two parameters: the element that the directive was called on, and an `attr` object associated with that element.

Note: You do not currently have the ability to access scope variables from the `templateUrl`function, since the template is requested before the scope is initialized.

Code: test_31.html, test_31.js

Note: When you create a directive, it is restricted to attribute and elements only by default. In order to create directives that are triggered by class name, you need to use the `restrict` option.
The `restrict` option is typically set to:

- `'A'` - only matches attribute name
- `'E'` - only matches element name
- `'C'` - only matches class name
- `'M'` - only matches comment

These restrictions can all be combined as needed:

- `'AEC'` - matches either attribute or element or class name

Let's change our directive to use `restrict: 'E'`:

Code: test_32.html, test_32.js

For more on the `restrict` property, see the API docs.

When should I use an attribute versus an element? Use an element when you are creating a component that is in control of the template. The common case for this is when you are creating a Domain-Specific Language for parts of your template. Use an attribute when you are decorating an existing element with new functionality.
Using an element for the `myCustomer` directive is clearly the right choice because you're not decorating an element with some "customer" behavior; you're defining the core behavior of the element as a customer component.

Isolating the Scope of a Directive

Our `myCustomer` directive above is great, but it has a fatal flaw. We can only use it once within a given scope.

In its current implementation, we'd need to create a different controller each time in order to re-use such a directive:

Code: test_33.html, test_33.js

This is clearly not a great solution.

What we want to be able to do is separate the scope inside a directive from the scope outside, and then map the outer scope to a directive's inner scope. We can do this by creating what we call an **isolate scope**. To do this, we can use a directive's `scope` option:

Code: test_34.html, test_34.js

Looking at `index.html`, the first `<my-customer>` element binds the `info` attribute to `naomi`, which we have exposed on our controller's scope. The second binds `info` to `igor`.

Let's take a closer look at the scope option:

```
//...
scope: {
  customerInfo: '=info'
},
//...
```

The **scope option** is an object that contains a property for each isolate scope binding. In this case it has just one property:

- Its name (`customerInfo`) corresponds to the directive's **isolate scope** property, `customerInfo`.
- Its value (`=info`) tells `$compile` to bind to the `info` attribute.

Note: These `=attr` attributes in the `scope` option of directives are normalized just like directive names. To bind to the attribute in `<div bind-to-this="thing">`, you'd specify a binding of `=bindToThis`.
For cases where the attribute name is the same as the value you want to bind to inside the directive's scope, you can use this shorthand syntax:

```
...
scope: {
  // same as '=customer'
  customer: '='
},
...
```

Besides making it possible to bind different data to the scope inside a directive, using an isolated scope has another effect.

We can show this by adding another property, `vojta`, to our scope and trying to access it from within our directive's template:

Code: test_35.html, test_35.js

Notice that `{{vojta.name}}` and `{{vojta.address}}` are empty, meaning they are undefined. Although we defined `vojta` in the controller, it's not available within the directive.

As the name suggests, the **isolate scope** of the directive isolates everything except models that you've explicitly added to the `scope: {}` hash object. This is helpful when building reusable components because it prevents a component from changing your model state except for the models that you explicitly pass in.

Note: Normally, a scope prototypically inherits from its parent. An isolated scope does not. See the "Directive Definition Object - scope" section for more information about isolate scopes.
Best Practice: Use the `scope` option to create isolate scopes when making components that you want to reuse throughout your app.

Creating a Directive that Manipulates the DOM

In this example we will build a directive that displays the current time. Once a second, it updates the DOM to reflect the current time.

Directives that want to modify the DOM typically use the `link` option to register DOM listeners as well as update the DOM. It is executed after the template has been cloned and is where directive logic will be put.

`link` takes a function with the following signature,`function link(scope, element, attrs, controller, transcludeFn) { ... }`, where:

- `scope` is an AngularJS scope object.
- `element` is the jqLite-wrapped element that this directive matches.
- `attrs` is a hash object with key-value pairs of normalized attribute names and their corresponding attribute values.
- `controller` is the directive's required controller instance(s) or its own controller (if any). The exact value depends on the directive's require property.
- `transcludeFn` is a transclude linking function pre-bound to the correct transclusion scope.

For more details on the `link` option refer to the `$compile` API page.
In our `link` function, we want to update the displayed time once a second, or whenever a user changes the time formatting string that our directive binds to. We will use the `$interval` service to call a handler on a regular basis. This is easier than using `$timeout` but also works better with end-to-end testing, where we want to ensure that all `$timeouts` have completed before completing the test. We also want to remove the `$interval` if the directive is deleted so we don't introduce a memory leak.

Code: test_36.html, test_36.js

There are a couple of things to note here. Just like the `module.controller` API, the function argument in `module.directive` is dependency injected. Because of this, we can use `$interval` and `dateFilter`inside our directive's `link` function.

We register an event `element.on('$destroy', ...)`. What fires this `$destroy` event?

There are a few special events that AngularJS emits. When a DOM node that has been compiled with AngularJS's compiler is destroyed, it emits a `$destroy` event. Similarly, when an AngularJS scope is destroyed, it broadcasts a `$destroy` event to listening scopes.

By listening to this event, you can remove event listeners that might cause memory leaks. Listeners registered to scopes and elements are automatically cleaned up when they are destroyed, but if you registered a listener on a service, or registered a listener on a DOM node that isn't being deleted, you'll have to clean it up yourself or you risk introducing a memory leak.

Best Practice: Directives should clean up after themselves. You can use`element.on('$destroy', ...)` or `scope.$on('$destroy', ...)` to run a clean-up function when the directive is removed.

Creating a Directive that Wraps Other Elements

We've seen that you can pass in models to a directive using the isolate scope, but sometimes it's desirable to be able to pass in an entire template rather than a string or an object. Let's say that we want to create a "dialog box" component. The dialog box should be able to wrap any arbitrary content.

To do this, we need to use the `transclude` option.

Code: test_37.html, test_37.js

What does this `transclude` option do, exactly? `transclude` makes the contents of a directive with this option have access to the scope **outside** of the directive rather than inside.

To illustrate this, see the example below. Notice that we've added a `link` function in `script.js` that redefines `name` as `Jeff`. What do you think the `{{name}}` binding will resolve to now?

Code: test_38.html, test_38.js

Ordinarily, we would expect that `{{name}}` would be `Jeff`. However, we see in this example that the `{{name}}` binding is still `Tobias`.

The `transclude` option changes the way scopes are nested. It makes it so that the **contents** of a transcluded directive have whatever scope is outside the directive, rather than whatever scope is on the inside. In doing so, it gives the contents access to the outside scope.

Note that if the directive did not create its own scope, then `scope` in `scope.name = 'Jeff'` would reference the outside scope and we would see `Jeff` in the output.

This behavior makes sense for a directive that wraps some content, because otherwise you'd have to pass in each model you wanted to use separately. If you have to pass in each model that you want to use, then you can't really have arbitrary contents, can you?

Best Practice: only use `transclude: true` when you want to create a directive that wraps arbitrary content.

Next, we want to add buttons to this dialog box, and allow someone using the directive to bind their own behavior to it.

Code: test_39.html, test_39.js

We want to run the function we pass by invoking it from the directive's scope, but have it run in the context of the scope where it's registered.

We saw earlier how to use `=attr` in the `scope` option, but in the above example, we're using `&attr`instead. The `&` binding allows a directive to trigger evaluation of an expression in the context of the original scope, at a specific time. Any legal expression is allowed, including an expression which contains a function call. Because of this, `&` bindings are ideal for binding callback functions to directive behaviors.

When the user clicks the `x` in the dialog, the directive's `close` function is called, thanks to `ng-click`.This call to `close` on the isolated scope actually evaluates the expression `hideDialog(message)` in the context of the original scope, thus running `Controller`'s `hideDialog` function.

Often it's desirable to pass data from the isolate scope via an expression to the parent scope, this can be done by passing a map of local variable names and values into the expression wrapper function. For example, the `hideDialog` function takes a message to display when the dialog is hidden. This is specified in the directive by calling `close({message: 'closing for now'})`. Then the local variable `message` will be available within the `on-close` expression.

Best Practice: use `&attr` in the `scope` option when you want your directive to expose an API for binding to behaviors.

Creating a Directive that Adds Event Listeners

Previously, we used the `link` function to create a directive that manipulated its DOM elements. Building upon that example, let's make a directive that reacts to events on its elements.

For instance, what if we wanted to create a directive that lets a user drag an element?

Code: test_40.html, test_40.js

Creating Directives that Communicate

You can compose any directives by using them within templates.

Sometimes, you want a component that's built from a combination of directives.

Imagine you want to have a container with tabs in which the contents of the container correspond to which tab is active.

Code: test_41.html, test_41.js

The `myPane` directive has a `require` option with value `^^myTabs`. When a directive uses this option, `$compile` will throw an error unless the specified controller is found. The `^^` prefix means that this directive searches for the controller on its parents. (A `^` prefix would make the directive look for the controller on its own element or its parents; without any prefix, the directive would look on its own element only.)

So where does this `myTabs` controller come from? Directives can specify controllers using the unsurprisingly named `controller` option. As you can see, the `myTabs` directive uses this option. Just like `ngController`, this option attaches a controller to the template of the directive.

If it is necessary to reference the controller or any functions bound to the controller from the template, you can use the option `controllerAs` to specify the name of the controller as an alias. The directive needs to define a scope for this configuration to be used. This is particularly useful in the case when the directive is used as a component.

Looking back at `myPane`'s definition, notice the last argument in its `link` function: `tabsCtrl`. When a directive requires a controller, it receives that controller as the fourth argument of its `link` function. Taking advantage of this, `myPane` can call the `addPane` function of `myTabs`.

If multiple controllers are required, the `require` option of the directive can take an array argument. The corresponding parameter being sent to the `link` function will also be an array.

```
angular.module('docsTabsExample', [])
.directive('myPane', function() {
  return {
    require: ['^^myTabs', 'ngModel'],
    restrict: 'E',
    transclude: true,
    scope: {
      title: '@'
    },
    link: function(scope, element, attrs, controllers) {
      var tabsCtrl = controllers[0],
          modelCtrl = controllers[1];


      tabsCtrl.addPane(scope);
    },
    templateUrl: 'my-pane.html'
  };
});
```

Savvy readers may be wondering what the difference is between `link` and `controller`. The basic difference is that `controller` can expose an API, and `link` functions can interact with controllers using `require`.

Best Practice: use `controller` when you want to expose an API to other directives. Otherwise use `link`.

Summary

Here we've seen the main use cases for directives. Each of these samples acts as a good starting point for creating your own directives.

You might also be interested in an in-depth explanation of the compilation process that's available in the compiler guide.

The `$compile` API page has a comprehensive list of directive options for reference.

controls for the purpose of grouping related controls together.

Section 13

Components

In AngularJS, a Component is a special kind of directive that uses a simpler configuration which is suitable for a component-based application structure.

This makes it easier to write an app in a way that's similar to using Web Components or using the new Angular's style of application architecture.

Advantages of Components:

- simpler configuration than plain directives
- promote sane defaults and best practices
- optimized for component-based architecture
- writing component directives will make it easier to upgrade to Angular

When not to use Components:

- for directives that need to perform actions in compile and pre-link functions, because they aren't available
- when you need advanced directive definition options like priority, terminal, multi-element
- when you want a directive that is triggered by an attribute or CSS class, rather than an element

Creating and configuring a Component

Components can be registered using the `.component()` method of an AngularJS module (returned by `angular.module()`). The method takes two arguments:

- The name of the Component (as string).
- The Component config object. (Note that, unlike the `.directive()` method, this method does **not**take a factory function.)

Code: test_42.html, test_42.js

It's also possible to add components via `$compileProvider` in a module's config phase.

Comparison between Directive definition and Component definition

	Directive	**Component**
bindings	No	Yes (binds to controller)
bindToController	Yes (default: false)	No (use bindings instead)
compile function	Yes	No
controller	Yes	Yes (default `function() {}`)
controllerAs	Yes (default: false)	Yes (default: `$ctrl`)
link functions	Yes	No
multiElement	Yes	No

	Directive	Component
priority	Yes	No
replace	Yes (deprecated)	No
require	Yes	Yes
restrict	Yes	No (restricted to elements only)
scope	Yes (default: false)	No (scope is always isolate)
template	Yes	Yes, injectable
templateNamespace	Yes	No
templateUrl	Yes	Yes, injectable
terminal	Yes	No
transclude	Yes (default: false)	Yes (default: false)

Component-based application architecture

As already mentioned, the component helper makes it easier to structure your application with a component-based architecture. But what makes a component beyond the options that the component helper has?

- **Components only control their own View and Data:** Components should never modify any data or DOM that is out of their own scope. Normally, in AngularJS it is possible to modify data anywhere in the application through scope inheritance and watches. This is practical, but can also lead to problems when it is not clear which part of the application is responsible for modifying the data. That is why component directives use an isolate scope, so a whole class of scope manipulation is not possible.

- **Components have a well-defined public API - Inputs and Outputs:** However, scope isolation only goes so far, because AngularJS uses two-way binding. So if you pass an object to a component like this - `bindings: {item: '='}`, and modify one of its properties, the change will be reflected in the parent component. For components however, only the component that owns the data should modify it, to make it easy to reason about what data is changed, and when. For that reason, components should follow a few simple conventions:
    - Inputs should be using `<` and `@` bindings. The `<` symbol denotes one-way bindings which are available since 1.5. The difference to `=` is that the bound properties in the component scope are not watched, which means if you assign a new value to the property in the component scope, it will not update the parent scope. Note however, that both parent and component scope reference the same object, so if you are changing object properties or array elements in the component, the parent will still reflect that change. The general rule should therefore be to never change an object or array property in the component scope. `@` bindings can be used when the input is a string, especially when the value of the binding doesn't change.

```
bindings: {
  hero: '<',
  comment: '@'
}
```

    - Outputs are realized with `&` bindings, which function as callbacks to component events.

```
bindings: {
  onDelete: '&',
  onUpdate: '&'
}
```

    - Instead of manipulating Input Data, the component calls the correct Output Event with the changed data. For a deletion, that means the component doesn't delete the `hero` itself, but sends it back to the owner component via the correct event.

```
<!-- note that we use kebab-case for bindings in the template as usual -->
<editable-field on-update="$ctrl.update('location', value)"></editable-field><br>
```

```
<button ng-click="$ctrl.onDelete({hero: $ctrl.hero})">Delete</button>
```

    - That way, the parent component can decide what to do with the event (e.g. delete an item or update the properties)

```
  ctrl.deleteHero(hero) {
    $http.delete(...).then(function() {
      var idx = ctrl.list.indexOf(hero);
      if (idx >= 0) {
        ctrl.list.splice(idx, 1);
      }
    });
```

```
}
```

- **Components have a well-defined lifecycle** Each component can implement "lifecycle hooks". These are methods that will be called at certain points in the life of the component. The following hook methods can be implemented:
    - `$onInit()` - Called on each controller after all the controllers on an element have been constructed and had their bindings initialized (and before the pre & post linking functions for the directives on this element). This is a good place to put initialization code for your controller.
    - `$onChanges(changesObj)` - Called whenever one-way bindings are updated. The `changesObj` is a hash whose keys are the names of the bound properties that have changed, and the values are an object of the form`{ currentValue, previousValue, isFirstChange() }`. Use this hook to trigger updates within a component such as cloning the bound value to prevent accidental mutation of the outer value.
    - `$doCheck()` - Called on each turn of the digest cycle. Provides an opportunity to detect and act on changes. Any actions that you wish to take in response to the changes that you detect must be invoked from this hook; implementing this has no effect on when `$onChanges` is called. For example, this hook could be useful if you wish to perform a deep equality check, or to check a Date object, changes to which would not be detected by AngularJS's change detector and thus not trigger `$onChanges`. This hook is invoked with no arguments; if detecting changes, you must store the previous value(s) for comparison to the current values.
    - `$onDestroy()` - Called on a controller when its containing scope is destroyed. Use this hook for releasing external resources, watches and event handlers.
    - `$postLink()` - Called after this controller's element and its children have been linked. Similar to the post-link function this hook can be used to set up DOM event handlers and do direct DOM manipulation. Note that child elements that contain `templateUrl` directives will not have been compiled and linked since they are waiting for their template to load asynchronously and their own compilation and linking has been suspended until that occurs. This hook can be considered analogous to the `ngAfterViewInit` and `ngAfterContentInit` hooks in Angular. Since the compilation process is rather different in AngularJS there is no direct mapping and care should be taken when upgrading.

By implementing these methods, your component can hook into its lifecycle.

- **An application is a tree of components:** Ideally, the whole application should be a tree of components that implement clearly defined inputs and outputs, and minimize two-way data binding. That way, it's easier to predict when data changes and what the state of a component is.

Example of a component tree

The following example expands on the simple component example and incorporates the concepts we introduced above:

Instead of an ngController, we now have a heroList component that holds the data of different heroes, and creates a heroDetail for each of them.

The heroDetail component now contains new functionality:

- a delete button that calls the bound `onDelete` function of the heroList component
- an input to change the hero location, in the form of a reusable editableField component. Instead of manipulating the hero object itself, it sends a changeset upwards to the heroDetail, which sends it upwards to the heroList component, which updates the original data.

Code: test_43.html, test_43.js

Components as route templates

Components are also useful as route templates (e.g. when using ngRoute). In a component-based application, every view is a component:

```
var myMod = angular.module('myMod', ['ngRoute']);
myMod.component('home', {
  template: '<h1>Home</h1><p>Hello, {{ $ctrl.user.name }} !</p>',
  controller: function() {
    this.user = {name: 'world'};
  }
});
myMod.config(function($routeProvider) {
  $routeProvider.when('/', {
    template: '<home></home>'
  });
```

```
});
```

When using $routeProvider, you can often avoid some boilerplate, by passing the resolved route dependencies directly to the component. Since 1.5, ngRoute automatically assigns the resolves to the route scope property `$resolve` (you can also configure the property name via `resolveAs`). When using components, you can take advantage of this and pass resolves directly into your component without creating an extra route controller:

```
var myMod = angular.module('myMod', ['ngRoute']);
myMod.component('home', {
  template: '<h1>Home</h1><p>Hello, {{ $ctrl.user.name }} !</p>',
  bindings: {
    user: '<'
  }
});
myMod.config(function($routeProvider) {
  $routeProvider.when('/', {
    template: '<home user="$resolve.user"></home>',
    resolve: {
      user: function($http) { return $http.get('...'); }
    }
  });
});
```

Intercomponent Communication

Directives can require the controllers of other directives to enable communication between each other. This can be achieved in a component by providing an object mapping for the `require` property. The object keys specify the property names under which the required controllers (object values) will be bound to the requiring component's controller.

Note that the required controllers will not be available during the instantiation of the controller, but they are guaranteed to be available just before the `$onInit` method is executed!

Here is a tab pane example built from components:

Code: test_44.html, test_44.js

Unit-testing Component Controllers

The easiest way to unit-test a component controller is by using the $componentController that is included in `ngMock`. The advantage of this method is that you do not have to create any DOM elements. The following example shows how to do this for the `heroDetail` component from above.

The examples use the Jasmine testing framework.

Controller Test:

```
describe('HeroDetailController', function() {
  var $componentController;

  beforeEach(module('heroApp'));
  beforeEach(inject(function(_$componentController_) {
    $componentController = _$componentController_;
  }));

  it('should call the `onDelete` binding, when deleting the hero', function()
{
    var onDeleteSpy = jasmine.createSpy('onDelete');
    var bindings = {hero: {}, onDelete: onDeleteSpy};
    var ctrl = $componentController('heroDetail', null, bindings);

    ctrl.delete();
    expect(onDeleteSpy).toHaveBeenCalledWith({hero: ctrl.hero});
  });

  it('should call the `onUpdate` binding, when updating a property',
function() {
    var onUpdateSpy = jasmine.createSpy('onUpdate');
    var bindings = {hero: {}, onUpdate: onUpdateSpy};
    var ctrl = $componentController('heroDetail', null, bindings);

    ctrl.update('foo', 'bar');
    expect(onUpdateSpy).toHaveBeenCalledWith({
      hero: ctrl.hero,
```

```
      prop: 'foo',
      value: 'bar'
    });
  });

});
```

Section 14

Component Router

Deprecation Notice: In an effort to keep synchronized with router changes in the new Angular, this implementation of the Component Router (ngComponentRouter module) has been deprecated and will not receive further updates. We are investigating backporting the new Angular Router to AngularJS, but alternatively, use the `ngRoute` module or community developed projects (e.g. ui-router).

Code: test_45.html, test_45.js

Section 15

Animations

AngularJS provides animation hooks for common directives such as ngRepeat, ngSwitch, and ngView, as well as custom directives via the `$animate` service. These animation hooks are set in place to trigger animations during the life cycle of various directives and when triggered, will attempt to perform a CSS Transition, CSS Keyframe Animation or a JavaScript callback Animation (depending on whether an animation is placed on the given directive). Animations can be placed using vanilla CSS by following the naming conventions set in place by AngularJS or with JavaScript code, defined as a factory.

Note that we have used non-prefixed CSS transition properties in our examples as the major browsers now support non-prefixed properties. If you intend to support older browsers or certain mobile browsers then you will need to include prefixed versions of the transition properties. Take a look at http://caniuse.com/#feat=css-transitions for what browsers require prefixes, andhttps://github.com/postcss/autoprefixer for a tool that can automatically generate the prefixes for you.

Animations are not available unless you include the `ngAnimate` module as a dependency of your application.

Below is a quick example of animations being enabled for `ngShow` and `ngHide`:

Code: test_46.html, test_46.js

Installation

See the API docs for `ngAnimate` for instructions on installing the module.

You may also want to setup a separate CSS file for defining CSS-based animations.

How they work

Animations in AngularJS are completely based on CSS classes. As long as you have a CSS class attached to an HTML element within your application, you can apply animations to it. Let's say for example that we have an HTML template with a repeater like so:

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

As you can see, the `repeated-item` class is present on the element that will be repeated and this class will be used as a reference within our application's CSS and/or JavaScript animation code to tell AngularJS to perform an animation.

As `ngRepeat` does its thing, each time a new item is added into the list, `ngRepeat` will add an`ng-enter` class to the element that is being added. When removed it will apply an `ng-leave` class and when moved around it will apply an `ng-move` class.

Taking a look at the following CSS code, we can see some transition and keyframe animation code set up for each of those events that occur when `ngRepeat` triggers them:

```
/*
  We are using CSS transitions for when the enter and move events
  are triggered for the element that has the `repeated-item` class
*/
.repeated-item.ng-enter, .repeated-item.ng-move {
  transition: all 0.5s linear;
  opacity: 0;
}
```

```
/*
  `.ng-enter-active` and `.ng-move-active` are where the transition
destination
  properties are set so that the animation knows what to animate
*/
.repeated-item.ng-enter.ng-enter-active,
.repeated-item.ng-move.ng-move-active {
  opacity: 1;
}

/*
  We are using CSS keyframe animations for when the `leave` event
  is triggered for the element that has the `repeated-item` class
*/
.repeated-item.ng-leave {
  animation: 0.5s my_animation;
}

@keyframes my_animation {
  from { opacity: 1; }
  to   { opacity: 0; }
}
```

The same approach to animation can be used using JavaScript code (**for simplicity, we rely on jQuery to perform animations here**):

```
myModule.animation('.repeated-item', function() {
  return {
    enter: function(element, done) {
      // Initialize the element's opacity
      element.css('opacity', 0);

      // Animate the element's opacity
      // (`element.animate()` is provided by jQuery)
```

```
      element.animate({opacity: 1}, done);

      // Optional `onDone`/`onCancel` callback function
      // to handle any post-animation cleanup operations
      return function(isCancelled) {
        if (isCancelled) {
          // Abort the animation if cancelled
          // (`element.stop()` is provided by jQuery)
          element.stop();
        }
      };
    },
    leave: function(element, done) {
      // Initialize the element's opacity
      element.css('opacity', 1);

      // Animate the element's opacity
      // (`element.animate()` is provided by jQuery)
      element.animate({opacity: 0}, done);

      // Optional `onDone`/`onCancel` callback function
      // to handle any post-animation cleanup operations
      return function(isCancelled) {
        if (isCancelled) {
          // Abort the animation if cancelled
          // (`element.stop()` is provided by jQuery)
          element.stop();
        }
      };
    },

    // We can also capture the following animation events:
    move: function(element, done) {},
```

```
    addClass: function(element, className, done) {},
    removeClass: function(element, className, done) {}
  }
});
```

With these generated CSS class names present on the element at the time, AngularJS automatically figures out whether to perform a CSS and/or JavaScript animation. Note that you can't have both CSS and JavaScript animations based on the same CSS class. See here for more details.

Class and `ngClass` animation hooks

AngularJS also pays attention to CSS class changes on elements by triggering the **add** and **remove**hooks. This means that if a CSS class is added to or removed from an element then an animation can be executed in between, before the CSS class addition or removal is finalized. (Keep in mind that AngularJS will only be able to capture class changes if an **interpolated expression** or the **ng-class** directive is used on the element.)

The example below shows how to perform animations during class changes:

Code: test_47.html, test_47.js

Although the CSS is a little different than what we saw before, the idea is the same.

Which directives support animations?

A handful of common AngularJS directives support and trigger animation hooks whenever any major event occurs during their life cycle. The table below explains in detail which animation events are triggered:

Directive	Supported Animations
ngRepeat	enter, leave, and move
ngIf	enter and leave

Directive	Supported Animations
ngSwitch	enter and leave
ngInclude	enter and leave
ngView	enter and leave
ngMessage / ngMessageExp	enter and leave
ngClass / {{class}}	add and remove
ngClassEven	add and remove
ngClassOdd	add and remove
ngHide	add and remove (the `ng-hide` class)
ngShow	add and remove (the `ng-hide` class)
ngModel	add and remove (various classes)
form / ngForm	add and remove (various classes)
ngMessages	add and remove (the `ng-active`/`ng-inactive` classes)

For a full breakdown of the steps involved during each animation event, refer to the API docs.

How do I use animations in my own directives?

Animations within custom directives can also be established by injecting `$animate` directly into your directive and making calls to it when needed.

```
myModule.directive('my-directive', ['$animate', function($animate) {
  return function(scope, element) {
    element.on('click', function() {
      if (element.hasClass('clicked')) {
        $animate.removeClass(element, 'clicked');
      } else {
        $animate.addClass(element, 'clicked');
      }
    });
  };
}]);
```

Animations on app bootstrap / page load

By default, animations are disabled when the AngularJS app bootstraps. If you are using the `ngApp`directive, this happens in the `DOMContentLoaded` event, so immediately after the page has been loaded. Animations are disabled, so that UI and content are instantly visible. Otherwise, with many animations on the page, the loading process may become too visually overwhelming, and the performance may suffer.

Internally, `ngAnimate` waits until all template downloads that are started right after bootstrap have finished. Then, it waits for the currently running $digest and one more after that, to finish. This ensures that the whole app has been compiled fully before animations are attempted.

If you do want your animations to play when the app bootstraps, you can enable animations globally in your main module's run function:

```
myModule.run(function($animate) {
  $animate.enabled(true);
});
```

How to (selectively) enable, disable and skip animations

There are several different ways to disable animations, both globally and for specific animations. Disabling specific animations can help to speed up the render performance, for example for

large `ngRepeat` lists that don't actually have animations. Because `ngAnimate` checks at runtime if animations are present, performance will take a hit even if an element has no animation.

During the config: $animateProvider.customFilter()

This function can be called during the config phase of an app. It takes a filter function as the only argument, which will then be used to "filter" animations (based on the animated element, the event type, and the animation options). Only when the filter function returns `true`, will the animation be performed. This allows great flexibility - you can easily create complex rules, such as allowing specific events only or enabling animations on specific subtrees of the DOM, and dynamically modify them, for example disabling animations at certain points in time or under certain circumstances.

```
app.config(function($animateProvider) {
  $animateProvider.customFilter(function(node, event, options) {
    // Example: Only animate `enter` and `leave` operations.
    return event === 'enter' || event === 'leave';
  });
});
```

The `customFilter` approach generally gives a big speed boost compared to other strategies, because the matching is done before other animation disabling strategies are checked.

Best Practice: Keep the filtering function as lean as possible, because it will be called for each DOM action (e.g. insertion, removal, class change) performed by "animation-aware" directives. See here for a list of built-in directives that support animations. Performing computationally expensive or time-consuming operations on each call of the filtering function can make your animations sluggish.

During the config: $animateProvider.classNameFilter()

This function too can be called during the config phase of an app. It takes a regex as the only argument, which will then be matched against the classes of any element that is about to be animated. The regex allows a lot of flexibility - you can either allow animations for specific classes only (useful when you are working with 3rd party animations), or exclude specific classes from getting animated.

```
app.config(function($animateProvider) {
  $animateProvider.classNameFilter(/animate-/);
});
/* prefixed with `animate-` */
.animate-fade-add.animate-fade-add-active {
  transition: all 1s linear;
  opacity: 0;
```

```
}
```

The `classNameFilter` approach generally gives a big speed boost compared to other strategies, because the matching is done before other animation disabling strategies are checked. However, that also means it is not possible to override class name matching with the two following strategies. It's of course still possible to enable / disable animations by changing an element's class name at runtime.

At runtime: $animate.enabled()

This function can be used to enable / disable animations in two different ways:

With a single `boolean` argument, it enables / disables animations globally: `$animate.enabled(false)` disables all animations in your app.

When the first argument is a native DOM or jqLite/jQuery element, the function enables / disables animations on this element *and all its children*: `$animate.enabled(myElement, false)`. You can still use it to re-enable animations for a child element, even if you have disabled them on a parent element. And compared to the `classNameFilter`, you can change the animation status at runtime instead of during the config phase.

Note however that the `$animate.enabled()` state for individual elements does not overwrite disabling rules that have been set in the classNameFilter.

Via CSS styles: overwriting styles in the `ng-animate` CSS class

Whenever an animation is started, `ngAnimate` applies the `ng-animate` class to the element for the whole duration of the animation. By applying CSS transition / animation styling to that class, you can skip an animation:

```
.my-class {
  transition: transform 2s;
}

.my-class:hover {
  transform: translateX(50px);
}

my-class.ng-animate {
  transition: 0s;
}
```

By setting `transition: 0s`, `ngAnimate` will ignore the existing transition styles, and not try to animate them (Javascript animations will still execute, though). This can be used to prevent issues with existing animations interfering with `ngAnimate`.

Preventing flicker before an animation starts

When nesting elements with structural animations, such as `ngIf`, into elements that have class-based animations such as `ngClass`, it sometimes happens that before the actual animation starts, there is a brief flicker or flash of content where the animated element is briefly visible.

To prevent this, you can apply styles to the `ng-[event]-prepare` class, which is added as soon as an animation is initialized, but removed before the actual animation starts (after waiting for a `$digest`). This class is only added for *structural* animations (`enter`, `move`, and `leave`).

Here's an example where you might see flickering:

```
<div ng-class="{red: myProp}">
  <div ng-class="{blue: myProp}">
    <div class="message" ng-if="myProp"></div>
  </div>
</div>
```

It is possible that during the `enter` event, the `.message` div will be briefly visible before it starts animating. In that case, you can add styles to the CSS that make sure the element stays hidden before the animation starts:

```
.message.ng-enter-prepare {
  opacity: 0;
}

/* Other animation styles ... */
```

Preventing collisions with existing animations and third-party libraries

By default, any `ngAnimate`-enabled directives will assume that `transition` / `animation` styles on the element are part of an `ngAnimate` animation. This can lead to problems when the styles are actually for animations that are independent of `ngAnimate`.

For example, an element acts as a loading spinner. It has an infinite css animation on it, and also an `ngIf`directive, for which no animations are defined:

```
.spinner {
  animation: rotating 2s linear infinite;
}


@keyframes rotating {
  from { transform: rotate(0deg); }
  to { transform: rotate(360deg); }
}
```

Now, when the `ngIf` expression changes, `ngAnimate` will see the spinner animation and use it to animate the `enter`/`leave` event, which doesn't work because the animation is infinite. The element will still be added / removed after a timeout, but there will be a noticeable delay.

This might also happen because some third-party frameworks place animation duration defaults across many element or className selectors in order to make their code small and reusable.

You can prevent this unwanted behavior by adding CSS to the `.ng-animate` class, that is added for the whole duration of each animation. Simply overwrite the transition / animation duration. In the case of the spinner, this would be:

```
.spinner.ng-animate {
  animation: 0s none;
  transition: 0s none;
}
```

If you do have CSS transitions / animations defined for the animation events, make sure they have a higher priority than any styles that are not related to `ngAnimate`.

You can also use one of the other strategies to disable animations.

Enable animations for elements outside of the AngularJS application DOM tree: $animate.pin()

Before animating, `ngAnimate` checks if the animated element is inside the application DOM tree. If not, no animation is run. Usually, this is not a problem since most apps use the `html` or `body` elements as their root.

Problems arise when the application is bootstrapped on a different element, and animations are attempted on elements that are outside the application tree, e.g. when libraries append popup or modal elements to the body tag.

You can use `$animate.pin(element, parentHost)` to associate an element with another element that belongs to your application. Simply call it before the element is added to the DOM / before the

animation starts, with the element you want to animate, and the element which should be its assumed parent.

More about animations

For a full breakdown of each method available on `$animate`, see the API documentation.

To see a complete demo, see the animation step in the phonecat tutorial.

Section 16

Modules

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.

AngularJS 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?

Code: test_48.html, test_48.js

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.

You can find a community style guide to help yourself when application grows.

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

Code: test_49.html, test_49.js

Module Loading

A module is a collection of providers, services, directives etc., and optionally config and run blocks which get applied to the application during the bootstrap process.

The module API describes all the available methods and how they can be used.

See Using Dependency Injection to find out which dependencies can be injected in each method.

Dependencies and Order of execution

Modules can list other modules as their dependencies. Depending on a module implies that the required module will be loaded before the requiring module is loaded.

In a single module the order of execution is as follows:

1. provider functions are executed, so they and the services they define can be made available to the $injector.
2. After that, the configuration blocks (config functions) are executed. This means the configuration blocks of the required modules execute before the configuration blocks of any requiring module.

This continues until all module dependencies has been resolved.

Then, the run blocks that have been collected from each module are executed in order of requirement.

Note: each module is only loaded once, even if multiple other modules require it. Note: the factory function for "values" and "services" is called lazily when the value/service is injected for the first time.

Registration in the config block

While it is recommended to register injectables directly with the module API, it is also possible to register services, directives etc. by injecting $provide or the individual service providers into the config function:

```
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', ...);
```

```
});
```

Run Blocks

Run blocks are the closest thing in AngularJS 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.

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 AngularJS. 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 AngularJS 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!');
    });
  });
});
```

Section 17

HTML Compiler

Overview

AngularJS's HTML compiler allows the developer to teach the browser new HTML syntax. The compiler allows you to attach behavior to any HTML element or attribute and even create new HTML elements or attributes with custom behavior. AngularJS calls these behavior extensions directives.

HTML has a lot of constructs for formatting the HTML for static documents in a declarative fashion. For example if something needs to be centered, there is no need to provide instructions to the browser how the window size needs to be divided in half so that the center is found, and that this

center needs to be aligned with the text's center. Simply add an `align="center"` attribute to any element to achieve the desired behavior. Such is the power of declarative language.

However, the declarative language is also limited, as it does not allow you to teach the browser new syntax. For example, there is no easy way to get the browser to align the text at 1/3 the position instead of 1/2. What is needed is a way to teach the browser new HTML syntax.

AngularJS comes pre-bundled with common directives which are useful for building any app. We also expect that you will create directives that are specific to your app. These extensions become a Domain Specific Language for building your application.

All of this compilation takes place in the web browser; no server side or pre-compilation step is involved.

Compiler

Compiler is an AngularJS service which traverses the DOM looking for attributes. The compilation process happens in two phases.

1. **Compile:** traverse the DOM and collect all of the directives. The result is a linking function.
2. **Link:** combine the directives with a scope and produce a live view. Any changes in the scope model are reflected in the view, and any user interactions with the view are reflected in the scope model. This makes the scope model the single source of truth.

Some directives such as `ng-repeat` clone DOM elements once for each item in a collection. Having a compile and link phase improves performance since the cloned template only needs to be compiled once, and then linked once for each clone instance.

Directive

A directive is a behavior which should be triggered when specific HTML constructs are encountered during the compilation process. The directives can be placed in element names, attributes, class names, as well as comments. Here are some equivalent examples of invoking the `ng-bind` directive.

```
<span ng-bind="exp"></span>
<span class="ng-bind: exp;"></span>
<ng-bind></ng-bind>
<!-- directive: ng-bind exp -->
```

A directive is just a function which executes when the compiler encounters it in the DOM. See directive APIfor in-depth documentation on how to write directives.

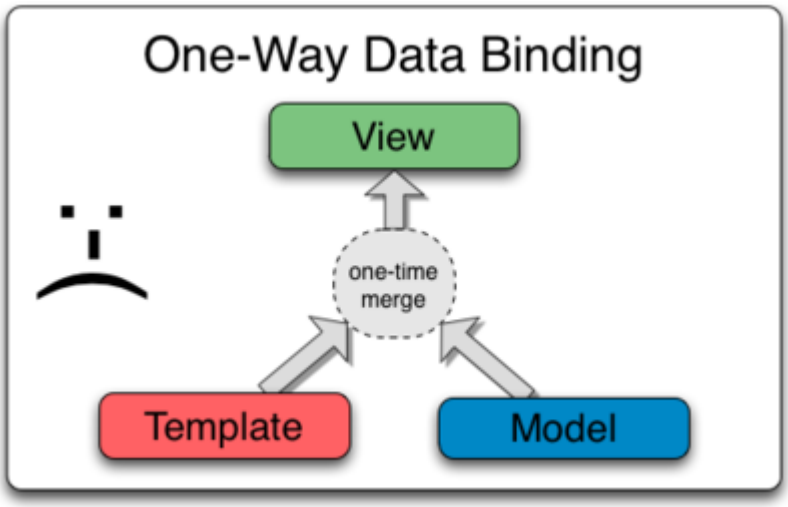
Here is a directive which makes any element draggable. Notice the `draggable` attribute on the `<span>`element.

Code: test_50.html, test_50.js

The presence of the `draggable` attribute on any element gives the element new behavior. We extended the vocabulary of the browser in a way which is natural to anyone who is familiar with the principles of HTML.

Understanding View

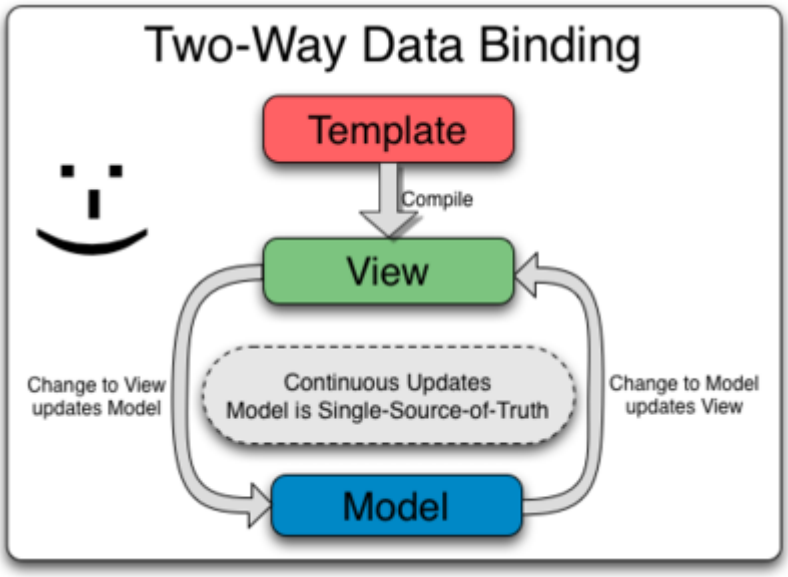
Most other templating systems consume a static string template and combine it with data, resulting in a new string. The resulting text is then `innerHTML`ed into an element.



This means that any changes to the data need to be re-merged with the template and then `innerHTML`ed into the DOM. Some of the issues with this approach are:

1. reading user input and merging it with data
2. clobbering user input by overwriting it
3. managing the whole update process
4. lack of behavior expressiveness

AngularJS is different. The AngularJS compiler consumes the DOM, not string templates. The result is a linking function, which when combined with a scope model results in a live view. The view and scope model bindings are transparent. The developer does not need to make any special calls to update the view. And because `innerHTML` is not used, you won't accidentally clobber user input. Furthermore, AngularJS directives can contain not just text bindings, but behavioral constructs as well.



The AngularJS approach produces a stable DOM. The DOM element instance bound to a model item instance does not change for the lifetime of the binding. This means that the code can get hold of the elements and register event handlers and know that the reference will not be destroyed by template data merge.

How directives are compiled

It's important to note that AngularJS operates on DOM nodes rather than strings. Usually, you don't notice this restriction because when a page loads, the web browser parses HTML into the DOM automatically.

HTML compilation happens in three phases:

1. `$compile` traverses the DOM and matches directives.

   If the compiler finds that an element matches a directive, then the directive is added to the list of directives that match the DOM element. A single element may match multiple directives.

2. Once all directives matching a DOM element have been identified, the compiler sorts the directives by their `priority`.

   Each directive's `compile` functions are executed. Each `compile` function has a chance to modify the DOM. Each `compile` function returns a `link` function. These functions are composed into a "combined" link function, which invokes each directive's returned `link` function.

3. `$compile` links the template with the scope by calling the combined linking function from the previous step. This in turn will call the linking function of the individual directives, registering listeners on the elements and setting up `$watch`s with the `scope` as each directive is configured to do.

The result of this is a live binding between the scope and the DOM. So at this point, a change in a model on the compiled scope will be reflected in the DOM.

Below is the corresponding code using the `$compile` service. This should help give you an idea of what AngularJS does internally.

```
var $compile = ...; // injected into your code

var scope = ...;

var parent = ...; // DOM element where the compiled template can be appended


var html = '<div ng-bind="exp"></div>';


// Step 1: parse HTML into DOM element

var template = angular.element(html);


// Step 2: compile the template

var linkFn = $compile(template);


// Step 3: link the compiled template with the scope.

var element = linkFn(scope);


// Step 4: Append to DOM (optional)

parent.appendChild(element);
```

The difference between Compile and Link

At this point you may wonder why the compile process has separate compile and link phases. The short answer is that compile and link separation is needed any time a change in a model causes a change in the **structure** of the DOM.

It's rare for directives to have a **compile function**, since most directives are concerned with working with a specific DOM element instance rather than changing its overall structure.

Directives often have a **link function**. A link function allows the directive to register listeners to the specific cloned DOM element instance as well as to copy content into the DOM from the scope.

Best Practice: Any operation which can be shared among the instance of directives should be moved to the compile function for performance reasons.

An Example of "Compile" Versus "Link"

To understand, let's look at a real-world example with `ngRepeat`:

```
Hello {{user.name}}, you have these actions:
<ul>
  <li ng-repeat="action in user.actions">
    {{action.description}}
  </li>
</ul>
```

When the above example is compiled, the compiler visits every node and looks for directives.

`{{user.name}}` matches the interpolation directive and `ng-repeat` matches the `ngRepeat` directive.

But ngRepeat has a dilemma.

It needs to be able to clone new `<li>` elements for every `action` in `user.actions`. This initially seems trivial, but it becomes more complicated when you consider that `user.actions` might have items added to it later. This means that it needs to save a clean copy of the `<li>` element for cloning purposes.

As new `actions` are inserted, the template `<li>` element needs to be cloned and inserted into `ul`. But cloning the `<li>` element is not enough. It also needs to compile the `<li>` so that its directives, like `{{action.description}}`, evaluate against the right scope.

A naive approach to solving this problem would be to simply insert a copy of the `<li>` element and then compile it. The problem with this approach is that compiling on every `<li>` element that we clone would duplicate a lot of the work. Specifically, we'd be traversing `<li>` each time before cloning it to find the directives. This would cause the compilation process to be slower, in turn making applications less responsive when inserting new nodes.

The solution is to break the compilation process into two phases:

the **compile phase** where all of the directives are identified and sorted by priority, and a **linking phase**where any work which "links" a specific instance of the scope and the specific instance of an `<li>` is performed.

Note: *Link* means setting up listeners on the DOM and setting up `$watch` on the Scope to keep the two in sync.
`ngRepeat` works by preventing the compilation process from descending into the `<li>` element so it can make a clone of the original and handle inserting and removing DOM nodes itself.

Instead the `ngRepeat` directive compiles `<li>` separately. The result of the `<li>` element compilation is a linking function which contains all of the directives contained in the `<li>` element, ready to be attached to a specific clone of the `<li>` element.

At runtime the `ngRepeat` watches the expression and as items are added to the array it clones the `<li>`element, creates a new scope for the cloned `<li>` element and calls the link function on the cloned `<li>`.

Understanding How Scopes Work with Transcluded Directives

One of the most common use cases for directives is to create reusable components.

Below is a pseudo code showing how a simplified dialog component may work.

```
<div>
  <button ng-click="show=true">show</button>

  <dialog title="Hello {{username}}."
          visible="show"
          on-cancel="show = false"
          on-ok="show = false; doSomething()">
     Body goes here: {{username}} is {{title}}.
  </dialog>
</div>
```

Clicking on the "show" button will open the dialog. The dialog will have a title, which is data bound to `username`, and it will also have a body which we would like to transclude into the dialog.

Here is an example of what the template definition for the `dialog` widget may look like.

```
<div ng-show="visible">
  <h3>{{title}}</h3>
  <div class="body" ng-transclude></div>
  <div class="footer">
    <button ng-click="onOk()">Save changes</button>
    <button ng-click="onCancel()">Close</button>
  </div>
</div>
```

This will not render properly, unless we do some scope magic.

The first issue we have to solve is that the dialog box template expects `title` to be defined. But we would like the template's scope property `title` to be the result of interpolating the `<dialog>` element's `title` attribute (i.e. `"Hello {{username}}"`). Furthermore, the buttons expect the `onOk` and `onCancel` functions to be present in the scope. This limits the usefulness of the widget. To solve the mapping issue we use the `scope` to create local variables which the template expects as follows:

```
scope: {
  title: '@',             // the title uses the data-binding from the parent
scope
  onOk: '&',              // create a delegate onOk function
  onCancel: '&',          // create a delegate onCancel function
  visible: '='            // set up visible to accept data-binding
```

```
}
```

Creating local properties on widget scope creates two problems:

1. isolation - if the user forgets to set `title` attribute of the dialog widget the dialog template will bind to parent scope property. This is unpredictable and undesirable.
2. transclusion - the transcluded DOM can see the widget locals, which may overwrite the properties which the transclusion needs for data-binding. In our example the `title` property of the widget clobbers the `title` property of the transclusion.

To solve the issue of lack of isolation, the directive declares a new `isolated` scope. An isolated scope does not prototypically inherit from the parent scope, and therefore we don't have to worry about accidentally clobbering any properties.

However `isolated` scope creates a new problem: if a transcluded DOM is a child of the widget isolated scope then it will not be able to bind to anything. For this reason the transcluded scope is a child of the original scope, before the widget created an isolated scope for its local variables. This makes the transcluded and widget isolated scope siblings.

This may seem to be unexpected complexity, but it gives the widget user and developer the least surprise.

Therefore the final directive definition looks something like this:

```
transclude: true,
scope: {
    title: '@',             // the title uses the data-binding from the
parent scope
    onOk: '&',              // create a delegate onOk function
    onCancel: '&',          // create a delegate onCancel function
    visible: '='            // set up visible to accept data-binding
},
restrict: 'E',
replace: true
```

Double Compilation, and how to avoid it

Double compilation occurs when an already compiled part of the DOM gets compiled again. This is an undesired effect and can lead to misbehaving directives, performance issues, and memory leaks. A common scenario where this happens is a directive that calls `$compile` in a directive link function on the directive element. In the following **faulty example**, a directive adds a mouseover behavior to a button with `ngClick` on it:

```
angular.module('app').directive('addMouseover', function($compile) {
  return {
```

```
    link: function(scope, element, attrs) {
      var newEl = angular.element('<span ng-show="showHint"> My
Hint</span>');
      element.on('mouseenter mouseleave', function() {
        scope.$apply('showHint = !showHint');
      });

      attrs.$set('addMouseover', null); // To stop infinite compile loop
      element.append(newEl);
      $compile(element)(scope); // Double compilation
    }
  }
})
```

At first glance, it looks like removing the original `addMouseover` attribute is all there is needed to make this example work. However, if the directive element or its children have other directives attached, they will be compiled and linked again, because the compiler doesn't keep track of which directives have been assigned to which elements.

This can cause unpredictable behavior, e.g. `ngClick` or other event handlers will be attached again. It can also degrade performance, as watchers for text interpolation are added twice to the scope.

Double compilation should therefore be avoided. In the above example, only the new element should be compiled:

```
angular.module('app').directive('addMouseover', function($compile) {
  return {
    link: function(scope, element, attrs) {
      var newEl = angular.element('<span ng-show="showHint"> My
Hint</span>');
      element.on('mouseenter mouseleave', function() {
        scope.$apply('showHint = !showHint');
      });

      element.append(newEl);
      $compile(newEl)(scope); // Only compile the new element
    }
  }
```

```
})
```

Another scenario is adding a directive programmatically to a compiled element and then executing compile again. See the following **faulty example**:

```
<input ng-model="$ctrl.value" add-options>
angular.module('app').directive('addOptions', function($compile) {
  return {
    link: function(scope, element, attrs) {
      attrs.$set('addOptions', null) // To stop infinite compile loop
      attrs.$set('ngModelOptions', '{debounce: 1000}');
      $compile(element)(scope); // Double compilation
    }
  }
});
```

In that case, it is necessary to intercept the *initial* compilation of the element:

1. Give your directive the `terminal` property and a higher priority than directives that should not be compiled twice. In the example, the compiler will only compile directives which have a priority of 100 or higher.
2. Inside this directive's compile function, add any other directive attributes to the template.
3. Compile the element, but restrict the maximum priority, so that any already compiled directives (including the `addOptions` directive) are not compiled again.
4. In the link function, link the compiled element with the element's scope.

```
angular.module('app').directive('addOptions', function($compile) {
  return {
    priority: 100, // ngModel has priority 1
    terminal: true,
    compile: function(templateElement, templateAttributes) {
      templateAttributes.$set('ngModelOptions', '{debounce: 1000}');

      // The third argument is the max priority. Only directives with
priority < 100 will be compiled,
      // therefore we don't need to remove the attribute
      var compiled = $compile(templateElement, null, 100);

      return function linkFn(scope) {
```

```
        compiled(scope) // Link compiled element to scope
      }
    }
  }
});
```

Section 18

Providers

Each web application you build is composed of objects that collaborate to get stuff done. These objects need to be instantiated and wired together for the app to work. In AngularJS apps most of these objects are instantiated and wired together automatically by the injector service.

The injector creates two types of objects, **services** and **specialized objects**.

Services are objects whose API is defined by the developer writing the service.

Specialized objects conform to a specific AngularJS framework API. These objects are one of controllers, directives, filters or animations.

The injector needs to know how to create these objects. You tell it by registering a "recipe" for creating your object with the injector. There are five recipe types.

The most verbose, but also the most comprehensive one is a Provider recipe. The remaining four recipe types — Value, Factory, Service and Constant — are just syntactic sugar on top of a provider recipe.

Let's take a look at the different scenarios for creating and using services via various recipe types. We'll start with the simplest case possible where various places in your code need a shared string and we'll accomplish this via Value recipe.

Note: A Word on Modules

In order for the injector to know how to create and wire together all of these objects, it needs a registry of "recipes". Each recipe has an identifier of the object and the description of how to create this object.

Each recipe belongs to an AngularJS module. An AngularJS module is a bag that holds one or more recipes. And since manually keeping track of module dependencies is no fun, a module can contain information about dependencies on other modules as well.

When an AngularJS application starts with a given application module, AngularJS creates a new instance of injector, which in turn creates a registry of recipes as a union of all recipes defined in the core "ng" module, application module and its dependencies. The injector then consults the recipe registry when it needs to create an object for your application.

Value Recipe

Let's say that we want to have a very simple service called "clientId" that provides a string representing an authentication id used for some remote API. You would define it like this:

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

myApp.value('clientId', 'a12345654321x');
```

Notice how we created an AngularJS module called `myApp`, and specified that this module definition contains a "recipe" for constructing the `clientId` service, which is a simple string in this case.

And this is how you would display it via AngularJS's data-binding:

```
myApp.controller('DemoController', ['clientId', function
DemoController(clientId) {

  this.clientId = clientId;

}]);

<html ng-app="myApp">

  <body ng-controller="DemoController as demo">

    Client ID: {{demo.clientId}}

  </body>

</html>
```

In this example, we've used the Value recipe to define the value to provide when `DemoController` asks for the service with id "clientId".

On to more complex examples!

Factory Recipe

The Value recipe is very simple to write, but lacks some important features we often need when creating services. Let's now look at the Value recipe's more powerful sibling, the Factory. The Factory recipe adds the following abilities:

- ability to use other services (have dependencies)

- service initialization
- delayed/lazy initialization

The Factory recipe constructs a new service using a function with zero or more arguments (these are dependencies on other services). The return value of this function is the service instance created by this recipe.

Note: All services in AngularJS are singletons. That means that the injector uses each recipe at most once to create the object. The injector then caches the reference for all future needs.

Since a Factory is a more powerful version of the Value recipe, the same service can be constructed with it. Using our previous `clientId` Value recipe example, we can rewrite it as a Factory recipe like this:

```
myApp.factory('clientId', function clientIdFactory() {
  return 'a12345654321x';
});
```

But given that the token is just a string literal, sticking with the Value recipe is still more appropriate as it makes the code easier to follow.

Let's say, however, that we would also like to create a service that computes a token used for authentication against a remote API. This token will be called `apiToken` and will be computed based on the `clientId` value and a secret stored in the browser's local storage:

```
myApp.factory('apiToken', ['clientId', function apiTokenFactory(clientId) {
  var encrypt = function(data1, data2) {
    // NSA-proof encryption algorithm:
    return (data1 + ':' + data2).toUpperCase();
  };

  var secret = window.localStorage.getItem('myApp.secret');
  var apiToken = encrypt(clientId, secret);

  return apiToken;
}]);
```

In the code above, we see how the `apiToken` service is defined via the Factory recipe that depends on the `clientId` service. The factory service then uses NSA-proof encryption to produce an authentication token.

Best Practice: name the factory functions as `<serviceId>Factory` (e.g., apiTokenFactory). While this naming convention is not required, it helps when navigating the codebase or looking at stack traces in the debugger.

Just like with the Value recipe, the Factory recipe can create a service of any type, whether it be a primitive, object literal, function, or even an instance of a custom type.

Service Recipe

JavaScript developers often use custom types to write object-oriented code. Let's explore how we could launch a unicorn into space via our `unicornLauncher` service which is an instance of a custom type:

```
function UnicornLauncher(apiToken) {


  this.launchedCount = 0;
  this.launch = function() {
    // Make a request to the remote API and include the apiToken
    ...
    this.launchedCount++;
  }
}
```

We are now ready to launch unicorns, but notice that UnicornLauncher depends on our `apiToken`. We can satisfy this dependency on `apiToken` using the Factory recipe:

```
myApp.factory('unicornLauncher', ["apiToken", function(apiToken) {
  return new UnicornLauncher(apiToken);
}]);
```

This is, however, exactly the use-case that the Service recipe is the most suitable for.

The Service recipe produces a service just like the Value or Factory recipes, but it does so by *invoking a constructor with the `new` operator*. The constructor can take zero or more arguments, which represent dependencies needed by the instance of this type.

Note: Service recipes follow a design pattern called constructor injection.

Since we already have a constructor for our UnicornLauncher type, we can replace the Factory recipe above with a Service recipe like this:

```
myApp.service('unicornLauncher', ["apiToken", UnicornLauncher]);
```

Much simpler!

Provider Recipe

As already mentioned in the intro, the Provider recipe is the core recipe type and all the other recipe types are just syntactic sugar on top of it. It is the most verbose recipe with the most abilities, but for most services it's overkill.

The Provider recipe is syntactically defined as a custom type that implements a `$get` method. This method is a factory function just like the one we use in the Factory recipe. In fact, if you define a Factory recipe, an empty Provider type with the `$get` method set to your factory function is automatically created under the hood.

You should use the Provider recipe only when you want to expose an API for application-wide configuration that must be made before the application starts. This is usually interesting only for reusable services whose behavior might need to vary slightly between applications.

Let's say that our `unicornLauncher` service is so awesome that many apps use it. By default the launcher shoots unicorns into space without any protective shielding. But on some planets the atmosphere is so thick that we must wrap every unicorn in tinfoil before sending it on its intergalactic trip, otherwise they would burn while passing through the atmosphere. It would then be great if we could configure the launcher to use the tinfoil shielding for each launch in apps that need it. We can make it configurable like so:

```
myApp.provider('unicornLauncher', function UnicornLauncherProvider() {
  var useTinfoilShielding = false;

  this.useTinfoilShielding = function(value) {
    useTinfoilShielding = !!value;
  };

  this.$get = ["apiToken", function unicornLauncherFactory(apiToken) {

    // let's assume that the UnicornLauncher constructor was also changed to
    // accept and use the useTinfoilShielding argument
    return new UnicornLauncher(apiToken, useTinfoilShielding);
  }];
});
```

To turn the tinfoil shielding on in our app, we need to create a config function via the module API and have the UnicornLauncherProvider injected into it:

```
myApp.config(["unicornLauncherProvider", function(unicornLauncherProvider) {
  unicornLauncherProvider.useTinfoilShielding(true);
}]);
```

Notice that the unicorn provider is injected into the config function. This injection is done by a provider injector which is different from the regular instance injector, in that it instantiates and wires (injects) all provider instances only.

During application bootstrap, before AngularJS goes off creating all services, it configures and instantiates all providers. We call this the configuration phase of the application life-cycle. During this phase, services aren't accessible because they haven't been created yet.

Once the configuration phase is over, interaction with providers is disallowed and the process of creating services starts. We call this part of the application life-cycle the run phase.

Constant Recipe

We've just learned how AngularJS splits the life-cycle into configuration phase and run phase and how you can provide configuration to your application via the config function. Since the config function runs in the configuration phase when no services are available, it doesn't have access even to simple value objects created via the Value recipe.

Since simple values, like URL prefixes, don't have dependencies or configuration, it's often handy to make them available in both the configuration and run phases. This is what the Constant recipe is for.

Let's say that our `unicornLauncher` service can stamp a unicorn with the planet name it's being launched from if this name was provided during the configuration phase. The planet name is application specific and is used also by various controllers during the runtime of the application. We can then define the planet name as a constant like this:

```
myApp.constant('planetName', 'Greasy Giant');
```

We could then configure the unicornLauncherProvider like this:

```
myApp.config(['unicornLauncherProvider', 'planetName',
function(unicornLauncherProvider, planetName) {
  unicornLauncherProvider.useTinfoilShielding(true);
  unicornLauncherProvider.stampText(planetName);
}]);
```

And since Constant recipe makes the value also available at runtime just like the Value recipe, we can also use it in our controller and template:

```
myApp.controller('DemoController', ["clientId", "planetName", function
DemoController(clientId, planetName) {
  this.clientId = clientId;
  this.planetName = planetName;
}]);
```

```
<html ng-app="myApp">
  <body ng-controller="DemoController as demo">
   Client ID: {{demo.clientId}}
   <br>
   Planet Name: {{demo.planetName}}
  </body>
</html>
```

Special Purpose Objects

Earlier we mentioned that we also have special purpose objects that are different from services. These objects extend the framework as plugins and therefore must implement interfaces specified by AngularJS. These interfaces are Controller, Directive, Filter and Animation.

The instructions for the injector to create these special objects (with the exception of the Controller objects) use the Factory recipe behind the scenes.

Let's take a look at how we would create a very simple component via the directive api that depends on the `planetName` constant we've just defined and displays the planet name, in our case: "Planet Name: Greasy Giant".

Since the directives are registered via the Factory recipe, we can use the same syntax as with factories.

```
myApp.directive('myPlanet', ['planetName', function
myPlanetDirectiveFactory(planetName) {
  // directive definition object
  return {
    restrict: 'E',
    scope: {},
    link: function($scope, $element) { $element.text('Planet: ' +
planetName); }
  }
}]);
```

We can then use the component like this:

```
<html ng-app="myApp">
  <body>
   <my-planet></my-planet>
```

```
  </body>
</html>
```

Using Factory recipes, you can also define AngularJS's filters and animations, but the controllers are a bit special. You create a controller as a custom type that declares its dependencies as arguments for its constructor function. This constructor is then registered with a module. Let's take a look at the `DemoController`, created in one of the early examples:

```
myApp.controller('DemoController', ['clientId', function
DemoController(clientId) {
  this.clientId = clientId;
}]);
```

The DemoController is instantiated via its constructor, every time the app needs an instance of DemoController (in our simple app it's just once). So unlike services, controllers are not singletons. The constructor is called with all the requested services, in our case the `clientId` service.

Conclusion

To wrap it up, let's summarize the most important points:

- The injector uses recipes to create two types of objects: services and special purpose objects
- There are five recipe types that define how to create objects: Value, Factory, Service, Provider and Constant.
- Factory and Service are the most commonly used recipes. The only difference between them is that the Service recipe works better for objects of a custom type, while the Factory can produce JavaScript primitives and functions.
- The Provider recipe is the core recipe type and all the other ones are just syntactic sugar on it.
- Provider is the most complex recipe type. You don't need it unless you are building a reusable piece of code that needs global configuration.
- All special purpose objects except for the Controller are defined via Factory recipes.

Features / Recipe type	Factory	Service	Value	Constant	Provider
can have dependencies	yes	yes	no	no	yes
uses type friendly injection	no	yes	yes*	yes*	no

Features / Recipe type	Factory	Service	Value	Constant	Provider
object available in config phase	no	no	no	yes	yes**
can create functions	yes	yes	yes	yes	yes
can create primitives	yes	no	yes	yes	yes

* at the cost of eager initialization by using `new` operator directly

** the service object is not available during the config phase, but the provider instance is (see the `unicornLauncherProvider` example above).

Section 19

Decorators

What are decorators?

Decorators are a design pattern that is used to separate modification or *decoration* of a class without modifying the original source code. In AngularJS, decorators are functions that allow a service, directive or filter to be modified prior to its usage.

How to use decorators

There are two ways to register decorators

- `$provide.decorator`, and
- `module.decorator`

Each provide access to a `$delegate`, which is the instantiated service/directive/filter, prior to being passed to the service that required it.

$provide.decorator

The decorator function allows access to a $delegate of the service once it has been instantiated. For example:

```
angular.module('myApp', [])

.config([ '$provide', function($provide) {

  $provide.decorator('$log', [
    '$delegate',
    function $logDecorator($delegate) {

      var originalWarn = $delegate.warn;
      $delegate.warn = function decoratedWarn(msg) {
        msg = 'Decorated Warn: ' + msg;
        originalWarn.apply($delegate, arguments);
      };

      return $delegate;
    }
  ]);
}]);
```

After the `$log` service has been instantiated the decorator is fired. The decorator function has a `$delegate` object injected to provide access to the service that matches the selector in the decorator. This `$delegate` will be the service you are decorating. The return value of the function *provided to the decorator* will take place of the service, directive, or filter being decorated.

The `$delegate` may be either modified or completely replaced. Given a service `myService` with a method `someFn`, the following could all be viable solutions:

Completely Replace the $delegate

```
angular.module('myApp', [])

.config([ '$provide', function($provide) {
```

```
  $provide.decorator('myService', [
    '$delegate',
    function myServiceDecorator($delegate) {

      var myDecoratedService = {
        // new service object to replace myService
      };
      return myDecoratedService;
    }
  ]);
}]);
```

Patch the $delegate

```
angular.module('myApp', [])

.config([ '$provide', function($provide) {

  $provide.decorator('myService', [
    '$delegate',
    function myServiceDecorator($delegate) {

      var someFn = $delegate.someFn;

      function aNewFn() {
        // new service function
        someFn.apply($delegate, arguments);
      }

      $delegate.someFn = aNewFn;
      return $delegate;
    }
  ]);
```

```
}]);
```

Augment the $delegate

```
angular.module('myApp', [])

.config([ '$provide', function($provide) {

  $provide.decorator('myService', [
    '$delegate',
    function myServiceDecorator($delegate) {

      function helperFn() {
        // an additional fn to add to the service
      }

      $delegate.aHelpfulAddition = helperFn;
      return $delegate;
    }
  ]);
}]);
```

Note that whatever is returned by the decorator function will replace that which is being decorated. For example, a missing return statement will wipe out the entire object being decorated.

Decorators have different rules for different services. This is because services are registered in different ways. Services are selected by name, however filters and directives are selected by appending `"Filter"` or `"Directive"` to the end of the name. The `$delegate` provided is dictated by the type of service.

Service Type	Selector	$delegate
Service	`serviceName`	The `object` or `function` returned by the service

Service Type	Selector	$delegate
Directive	`directiveName + 'Directive'`	An `Array.<DirectiveObject>`[1]
Filter	`filterName + 'Filter'`	The `function` returned by the filter

1. Multiple directives may be registered to the same selector/name

NOTE: Developers should take care in how and why they are modifying the `$delegate` for the service. Not only should expectations for the consumer be kept, but some functionality (such as directive registration) does not take place after decoration, but during creation/registration of the original service. This means, for example, that an action such as pushing a directive object to a directive `$delegate` will likely result in unexpected behavior. Furthermore, great care should be taken when decorating core services, directives, or filters as this may unexpectedly or adversely affect the functionality of the framework.

module.decorator

This function is the same as the `$provide.decorator` function except it is exposed through the module API. This allows you to separate your decorator patterns from your module config blocks.

Like with `$provide.decorator`, the `module.decorator` function runs during the config phase of the app. That means you can define a `module.decorator` before the decorated service is defined.

Since you can apply multiple decorators, it is noteworthy that decorator application always follows order of declaration:

- If a service is decorated by both `$provide.decorator` and `module.decorator`, the decorators are applied in order:

```
angular
.module('theApp', [])
.factory('theFactory', theFactoryFn)
.config(function($provide) {
  $provide.decorator('theFactory', provideDecoratorFn); // runs first
})
.decorator('theFactory', moduleDecoratorFn); // runs seconds
```

- If the service has been declared multiple times, a decorator will decorate the service that has been declared last:

```
angular
  .module('theApp', [])
```

```
  .factory('theFactory', theFactoryFn)
  .decorator('theFactory', moduleDecoratorFn)
  .factory('theFactory', theOtherFactoryFn);


// `theOtherFactoryFn` is selected as 'theFactory' provider and it is
decorated via `moduleDecoratorFn`.
```

Example Applications

The following sections provide examples each of a service decorator, a directive decorator, and a filter decorator.

Service Decorator Example

This example shows how we can replace the $log service with our own to display log messages.

Code: test_51.html, test_51.js

Directive Decorator Example

Failed interpolated expressions in `ng-href` attributes can easily go unnoticed. We can decorate `ngHref`to warn us of those conditions.

Code: test_52.html, test_52.js

Filter Decorator Example

Let's say we have created an app that uses the default format for many of our `Date` filters. Suddenly requirements have changed (that never happens) and we need all of our default dates to be `'shortDate'`instead of `'mediumDate'`.

Code: test_53.html, test_53.js

Section 20

Bootstrap

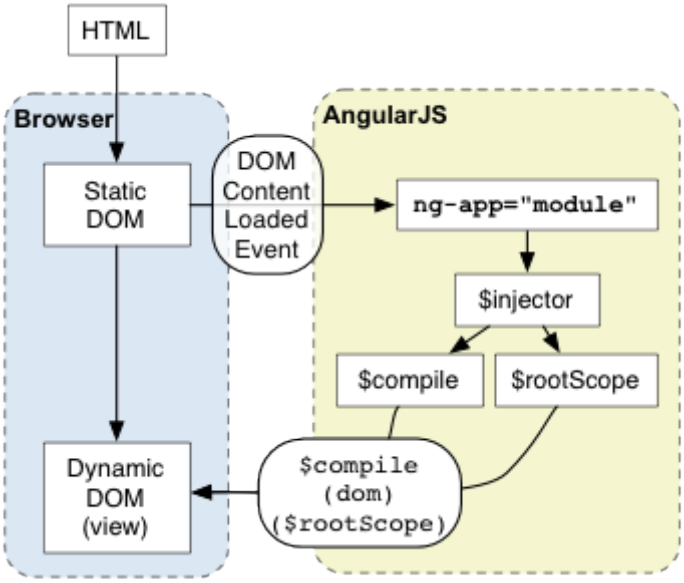
AngularJS `<script>` Tag

This example shows the recommended path for integrating AngularJS with what we call automatic initialization.

```
<!doctype html>
<html xmlns:ng="http://angularjs.org" ng-app>
  <body>
    ...
    <script src="angular.js"></script>
  </body>
</html>
```

1. Place the `script` tag at the bottom of the page. Placing script tags at the end of the page improves app load time because the HTML loading is not blocked by loading of the `angular.js` script. You can get the latest bits from http://code.angularjs.org. Please don't link your production code to this URL, as it will expose a security hole on your site. For experimental development linking to our site is fine.
    - Choose: `angular-[version].js` for a human-readable file, suitable for development and debugging.
    - Choose: `angular-[version].min.js` for a compressed and obfuscated file, suitable for use in production.
2. Place `ng-app` to the root of your application, typically on the `<html>` tag if you want AngularJS to auto-bootstrap your application.
3. If you choose to use the old style directive syntax `ng:` then include xml-namespace in `html` when running the page in the XHTML mode. (This is here for historical reasons, and we no longer recommend use of `ng:`.)

Automatic Initialization



AngularJS initializes automatically upon `DOMContentLoaded` event or when the `angular.js` script is evaluated if at that time `document.readyState` is set to `'complete'`. At this point AngularJS looks for the `ngApp` directive which designates your application root. If the `ngApp` directive is found then AngularJS will:

- load the module associated with the directive.
- create the application injector
- compile the DOM treating the `ngApp` directive as the root of the compilation. This allows you to tell it to treat only a portion of the DOM as an AngularJS application.

```
<!doctype html>
<html ng-app="optionalModuleName">
  <body>
    I can add: {{ 1+2 }}.
    <script src="angular.js"></script>
  </body>
</html>
```

As a best practice, consider adding an `ng-strict-di` directive on the same element as `ng-app`:

```
<!doctype html>
<html ng-app="optionalModuleName" ng-strict-di>
  <body>
    I can add: {{ 1+2 }}.
    <script src="angular.js"></script>
  </body>
```

```
</html>
```

This will ensure that all services in your application are properly annotated. See the dependency injection strict mode docs for more.

Manual Initialization

If you need to have more control over the initialization process, you can use a manual bootstrapping method instead. Examples of when you'd need to do this include using script loaders or the need to perform an operation before AngularJS compiles a page.

Here is an example of manually initializing AngularJS:

```
<!doctype html>
<html>
<body>
  <div ng-controller="MyController">
    Hello {{greetMe}}!
  </div>
  <script src="http://code.angularjs.org/snapshot/angular.js"></script>

  <script>
    angular.module('myApp', [])
      .controller('MyController', ['$scope', function ($scope) {
        $scope.greetMe = 'World';
      }]);

    angular.element(function() {
      angular.bootstrap(document, ['myApp']);
    });
  </script>
</body>
</html>
```

Note that we provided the name of our application module to be loaded into the injector as the second parameter of the `angular.bootstrap` function. Notice that `angular.bootstrap` will not

create modules on the fly. You must create any custom modulesbefore you pass them as a parameter.

You should call `angular.bootstrap()` *after* you've loaded or defined your modules. You cannot add controllers, services, directives, etc after an application bootstraps.

Note: You should not use the ng-app directive when manually bootstrapping your app.
This is the sequence that your code should follow:

1. After the page and all of the code is loaded, find the root element of your AngularJS application, which is typically the root of the document.
2. Call `angular.bootstrap` to compile the element into an executable, bi-directionally bound application.

Things to keep in mind

There are a few things to keep in mind regardless of automatic or manual bootstrapping:

- While it's possible to bootstrap more than one AngularJS application per page, we don't actively test against this scenario. It's possible that you'll run into problems, especially with complex apps, so caution is advised.
- Do not bootstrap your app on an element with a directive that uses transclusion, such as `ngIf`, `ngInclude` and `ngView`. Doing this misplaces the app `$rootElement` and the app's injector, causing animations to stop working and making the injector inaccessible from outside the app.

Deferred Bootstrap

This feature enables tools like Batarang and test runners to hook into angular's bootstrap process and sneak in more modules into the DI registry which can replace or augment DI services for the purpose of instrumentation or mocking out heavy dependencies.

If `window.name` contains prefix `NG_DEFER_BOOTSTRAP!` when `angular.bootstrap` is called, the bootstrap process will be paused until `angular.resumeBootstrap()` is called.

`angular.resumeBootstrap()` takes an optional array of modules that should be added to the original list of modules that the app was about to be bootstrapped with.

Section 21

Unit Testing

JavaScript is a dynamically typed language which comes with great power of expression, but it also comes with almost no help from the compiler. For this reason we feel very strongly that any code written in JavaScript needs to come with a strong set of tests. We have built many features into AngularJS which make testing your AngularJS applications easy. With AngularJS, there is no excuse for not testing.

Separation of Concerns

Unit testing, as the name implies, is about testing individual units of code. Unit tests try to answer questions such as "Did I think about the logic correctly?" or "Does the sort function order the list in the right order?"

In order to answer such a question it is very important that we can isolate the unit of code under test. That is because when we are testing the sort function we don't want to be forced into creating related pieces such as the DOM elements, or making any XHR calls to fetch the data to sort.

While this may seem obvious it can be very difficult to call an individual function on a typical project. The reason is that the developers often mix concerns resulting in a piece of code which does everything. It makes an XHR request, it sorts the response data, and then it manipulates the DOM.

With AngularJS, we try to make it easy for you to do the right thing. For your XHR requests, we provide dependency injection, so your requests can be simulated. For the DOM, we abstract it, so you can test your model without having to manipulate the DOM directly. Your tests can then assert that the data has been sorted without having to create or look at the state of the DOM or to wait for any XHR requests to return data. The individual sort function can be tested in isolation.

With great power comes great responsibility

AngularJS is written with testability in mind, but it still requires that you do the right thing. We tried to make the right thing easy, but if you ignore these guidelines you may end up with an untestable application.

Dependency Injection

AngularJS comes with dependency injection built-in, which makes testing components much easier, because you can pass in a component's dependencies and stub or mock them as you wish.

Components that have their dependencies injected allow them to be easily mocked on a test by test basis, without having to mess with any global variables that could inadvertently affect another test.

Additional tools for testing AngularJS applications

For testing AngularJS applications there are certain tools that you should use that will make testing much easier to set up and run.

Karma

Karma is a JavaScript command line tool that can be used to spawn a web server which loads your application's source code and executes your tests. You can configure Karma to run against a number of browsers, which is useful for being confident that your application works on all browsers you need to support. Karma is executed on the command line and will display the results of your tests on the command line once they have run in the browser.

Karma is a NodeJS application, and should be installed through npm/yarn. Full installation instructions are available on the Karma website.

Jasmine

Jasmine is a behavior driven development framework for JavaScript that has become the most popular choice for testing AngularJS applications. Jasmine provides functions to help with structuring your tests and also making assertions. As your tests grow, keeping them well structured and documented is vital, and Jasmine helps achieve this.

In Jasmine we use the `describe` function to group our tests together:

```
describe("sorting the list of users", function() {
  // individual tests go here
});
```

And then each individual test is defined within a call to the `it` function:

```
describe('sorting the list of users', function() {
  it('sorts in descending order by default', function() {
    // your test assertion goes here
```

```
  });
});
```

Grouping related tests within `describe` blocks and describing each individual test within an `it` call keeps your tests self documenting.

Finally, Jasmine provides matchers which let you make assertions:

```
describe('sorting the list of users', function() {
  it('sorts in descending order by default', function() {
    var users = ['jack', 'igor', 'jeff'];
    var sorted = sortUsers(users);
    expect(sorted).toEqual(['jeff', 'jack', 'igor']);
  });
});
```

Jasmine comes with a number of matchers that help you make a variety of assertions. You should read the Jasmine documentation to see what they are. To use Jasmine with Karma, we use the karma-jasmine test runner.

angular-mocks

AngularJS also provides the `ngMock` module, which provides mocking for your tests. This is used to inject and mock AngularJS services within unit tests. In addition, it is able to extend other modules so they are synchronous. Having tests synchronous keeps them much cleaner and easier to work with. One of the most useful parts of ngMock is `$httpBackend`, which lets us mock XHR requests in tests, and return sample data instead.

Testing a Controller

Because AngularJS separates logic from the view layer, it keeps controllers easy to test. Let's take a look at how we might test the controller below, which provides `$scope.grade`, which sets a property on the scope based on the length of the password.

```
angular.module('app', [])
.controller('PasswordController', function PasswordController($scope) {
  $scope.password = '';
  $scope.grade = function() {
    var size = $scope.password.length;
    if (size > 8) {
```

```
      $scope.strength = 'strong';
    } else if (size > 3) {
      $scope.strength = 'medium';
    } else {
      $scope.strength = 'weak';
    }
  };
});
```

Because controllers are not available on the global scope, we need to use `angular.mock.inject` to inject our controller first. The first step is to use the `module` function, which is provided by angular-mocks. This loads in the module it's given, so it is available in your tests. We pass this into `beforeEach`, which is a function Jasmine provides that lets us run code before each test. Then we can use `inject` to access `$controller`, the service that is responsible for instantiating controllers.

```
describe('PasswordController', function() {
  beforeEach(module('app'));


  var $controller, $rootScope;


  beforeEach(inject(function(_$controller_, _$rootScope_){
    // The injector unwraps the underscores (_) from around the parameter
names when matching
    $controller = _$controller_;
    $rootScope = _$rootScope_;
  }));


  describe('$scope.grade', function() {
    it('sets the strength to "strong" if the password length is >8 chars',
function() {
      var $scope = $rootScope.$new();
      var controller = $controller('PasswordController', { $scope: $scope });
      $scope.password = 'longerthaneightchars';
      $scope.grade();
      expect($scope.strength).toEqual('strong');
    });
```

```
  });
});
```

Notice how by nesting the `describe` calls and being descriptive when calling them with strings, the test is very clear. It documents exactly what it is testing, and at a glance you can quickly see what is happening. Now let's add the test for when the password is less than three characters, which should see `$scope.strength` set to "weak":

```
describe('PasswordController', function() {
  beforeEach(module('app'));


  var $controller;


  beforeEach(inject(function(_$controller_){
    // The injector unwraps the underscores (_) from around the parameter
names when matching
    $controller = _$controller_;
  }));


  describe('$scope.grade', function() {
    it('sets the strength to "strong" if the password length is >8 chars',
function() {
      var $scope = {};
      var controller = $controller('PasswordController', { $scope: $scope });
      $scope.password = 'longerthaneightchars';
      $scope.grade();
      expect($scope.strength).toEqual('strong');
    });


    it('sets the strength to "weak" if the password length <3 chars',
function() {
      var $scope = {};
      var controller = $controller('PasswordController', { $scope: $scope });
      $scope.password = 'a';
      $scope.grade();
      expect($scope.strength).toEqual('weak');
    });
```

```
  });
});
```

Now we have two tests, but notice the duplication between the tests. Both have to create the `$scope` variable and create the controller. As we add new tests, this duplication is only going to get worse. Thankfully, Jasmine provides `beforeEach`, which lets us run a function before each individual test. Let's see how that would tidy up our tests:

```
describe('PasswordController', function() {
  beforeEach(module('app'));


  var $controller;


  beforeEach(inject(function(_$controller_){
    // The injector unwraps the underscores (_) from around the parameter
names when matching
    $controller = _$controller_;
  }));


  describe('$scope.grade', function() {
    var $scope, controller;


    beforeEach(function() {
      $scope = {};
      controller = $controller('PasswordController', { $scope: $scope });
    });


    it('sets the strength to "strong" if the password length is >8 chars',
function() {
      $scope.password = 'longerthaneightchars';
      $scope.grade();
      expect($scope.strength).toEqual('strong');
    });


    it('sets the strength to "weak" if the password length <3 chars',
function() {
      $scope.password = 'a';
```

```
      $scope.grade();
      expect($scope.strength).toEqual('weak');
    });
  });
});
```

We've moved the duplication out and into the `beforeEach` block. Each individual test now only contains the code specific to that test, and not code that is general across all tests. As you expand your tests, keep an eye out for locations where you can use `beforeEach` to tidy up tests. `beforeEach` isn't the only function of this sort that Jasmine provides, and the documentation lists the others.

Testing Filters

Filters are functions which transform the data into a user readable format. They are important because they remove the formatting responsibility from the application logic, further simplifying the application logic.

```
myModule.filter('length', function() {
  return function(text) {
    return ('' + (text || '')).length;
  }
});

describe('length filter', function() {

  var $filter;

  beforeEach(inject(function(_$filter_){
    $filter = _$filter_;
  }));

  it('returns 0 when given null', function() {
    var length = $filter('length');
    expect(length(null)).toEqual(0);
  });
```

```
  it('returns the correct value when given a string of chars', function() {
    var length = $filter('length');
    expect(length('abc')).toEqual(3);
  });
});
```

Testing Directives

Directives in AngularJS are responsible for encapsulating complex functionality within custom HTML tags, attributes, classes or comments. Unit tests are very important for directives because the components you create with directives may be used throughout your application and in many different contexts.

Simple HTML Element Directive

Let's start with an AngularJS app with no dependencies.

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

Now we can add a directive to our app.

```
app.directive('aGreatEye', function () {
    return {
        restrict: 'E',
        replace: true,
        template: '<h1>lidless, wreathed in flame, {{1 + 1}} times</h1>'
    };
});
```

This directive is used as a tag `<a-great-eye></a-great-eye>`. It replaces the entire tag with the template `<h1>lidless, wreathed in flame, {{1 + 1}} times</h1>`. Now we are going to write a jasmine unit test to verify this functionality. Note that the expression `{{1 + 1}}` times will also be evaluated in the rendered content.

```
describe('Unit testing great quotes', function() {
  var $compile,
      $rootScope;
```

```
  // Load the myApp module, which contains the directive
  beforeEach(module('myApp'));

  // Store references to $rootScope and $compile
  // so they are available to all tests in this describe block
  beforeEach(inject(function(_$compile_, _$rootScope_){
    // The injector unwraps the underscores (_) from around the parameter
names when matching
    $compile = _$compile_;
    $rootScope = _$rootScope_;
  }));

  it('Replaces the element with the appropriate content', function() {
    // Compile a piece of HTML containing the directive
    var element = $compile("<a-great-eye></a-great-eye>")($rootScope);
    // fire all the watches, so the scope expression {{1 + 1}} will be
evaluated
    $rootScope.$digest();
    // Check that the compiled element contains the templated content
    expect(element.html()).toContain("lidless, wreathed in flame, 2 times");
  });
});
```

We inject the $compile service and $rootScope before each jasmine test. The $compile service is used to render the aGreatEye directive. After rendering the directive we ensure that the directive has replaced the content and "lidless, wreathed in flame, 2 times" is present.

Underscore notation: The use of the underscore notation (e.g.: `_$rootScope_`) is a convention wide spread in AngularJS community to keep the variable names clean in your tests. That's why the `$injector` strips out the leading and the trailing underscores when matching the parameters. The underscore rule applies ***only*** if the name starts **and** ends with exactly one underscore, otherwise no replacing happens.

Testing Transclusion Directives

Directives that use transclusion are treated specially by the compiler. Before their compile function is called, the contents of the directive's element are removed from the element and provided via a transclusion function. The directive's template is then appended to the directive's element, to which it can then insert the transcluded content into its template.

Before compilation:

```
<div transclude-directive>
  Some transcluded content
</div>
```

After transclusion extraction:

```
<div transclude-directive></div>
```

After compilation:

```
<div transclude-directive>
  Some Template
  <span ng-transclude>Some transcluded content</span>
</div>
```

If the directive is using 'element' transclusion, the compiler will actually remove the directive's entire element from the DOM and replace it with a comment node. The compiler then inserts the directive's template "after" this comment node, as a sibling.

Before compilation

```
<div element-transclude>
  Some Content
</div>
```

After transclusion extraction

```
<!-- elementTransclude -->
```

After compilation:

```
<!-- elementTransclude -->
<div element-transclude>
  Some Template
  <span ng-transclude>Some transcluded content</span>
</div>
```

It is important to be aware of this when writing tests for directives that use 'element' transclusion. If you place the directive on the root element of the DOM fragment that you pass to `$compile`, then the DOM node returned from the linking function will be the comment node and you will lose the ability to access the template and transcluded content.

```
var node = $compile('<div element-transclude></div>')($rootScope);
expect(node[0].nodeType).toEqual(node.COMMENT_NODE);
expect(node[1]).toBeUndefined();
```

To cope with this you simply ensure that your 'element' transclude directive is wrapped in an element, such as a `<div>`.

```
var node = $compile('<div><div element-transclude></div></div>')($rootScope);
var contents = node.contents();
expect(contents[0].nodeType).toEqual(node.COMMENT_NODE);
expect(contents[1].nodeType).toEqual(node.ELEMENT_NODE);
```

Testing Directives With External Templates

If your directive uses `templateUrl`, consider using karma-ng-html2js-preprocessor to pre-compile HTML templates and thus avoid having to load them over HTTP during test execution. Otherwise you may run into issues if the test directory hierarchy differs from the application's.

Testing Promises

When testing promises, it's important to know that the resolution of promises is tied to the digest cycle. That means a promise's `then`, `catch` and `finally` callback functions are only called after a digest has run. In tests, you can trigger a digest by calling a scope's `$apply` function. If you don't have a scope in your test, you can inject the $rootScope and call `$apply` on it. There is also an example of testing promises in the `$q` service documentation.

Using `beforeAll()`

Jasmine's `beforeAll()` and mocha's `before()` hooks are often useful for sharing test setup - either to reduce test run-time or simply to make for more focused test cases.

By default, ngMock will create an injector per test case to ensure your tests do not affect each other. However, if we want to use `beforeAll()`, ngMock will have to create the injector before any test cases are run, and share that injector through all the cases for that `describe`. That is where module.sharedInjector() comes in. When it's called within a `describe` block, a single injector is shared between all hooks and test cases run in that block.

In the example below we are testing a service that takes a long time to generate its answer. To avoid having all of the assertions we want to write in a single test case, module.sharedInjector() and Jasmine's `beforeAll()` are used to run the service only once. The test cases then all make assertions about the properties added to the service instance.

```
describe("Deep Thought", function() {


  module.sharedInjector();


  beforeAll(module("UltimateQuestion"));


  beforeAll(inject(function(DeepThought) {
    expect(DeepThought.answer).toBeUndefined();
    DeepThought.generateAnswer();
  }));


  it("has calculated the answer correctly", inject(function(DeepThought) {
    // Because of sharedInjector, we have access to the instance of the
DeepThought service
    // that was provided to the beforeAll() hook. Therefore we can test the
generated answer
    expect(DeepThought.answer).toBe(42);
  }));


  it("has calculated the answer within the expected time",
inject(function(DeepThought) {
    expect(DeepThought.runTimeMillennia).toBeLessThan(8000);
  }));


  it("has double checked the answer", inject(function(DeepThought) {
    expect(DeepThought.absolutelySureItIsTheRightAnswer).toBe(true);
  }));


});
```

Section 22

E2E Testing

As applications grow in size and complexity, it becomes unrealistic to rely on manual testing to verify the correctness of new features, catch bugs and notice regressions. Unit tests are the first line of defense for catching bugs, but sometimes issues come up with integration between components which can't be captured in a unit test. End-to-end tests are made to find these problems.

We have built Protractor, an end to end test runner which simulates user interactions that will help you verify the health of your AngularJS application.

Using Protractor

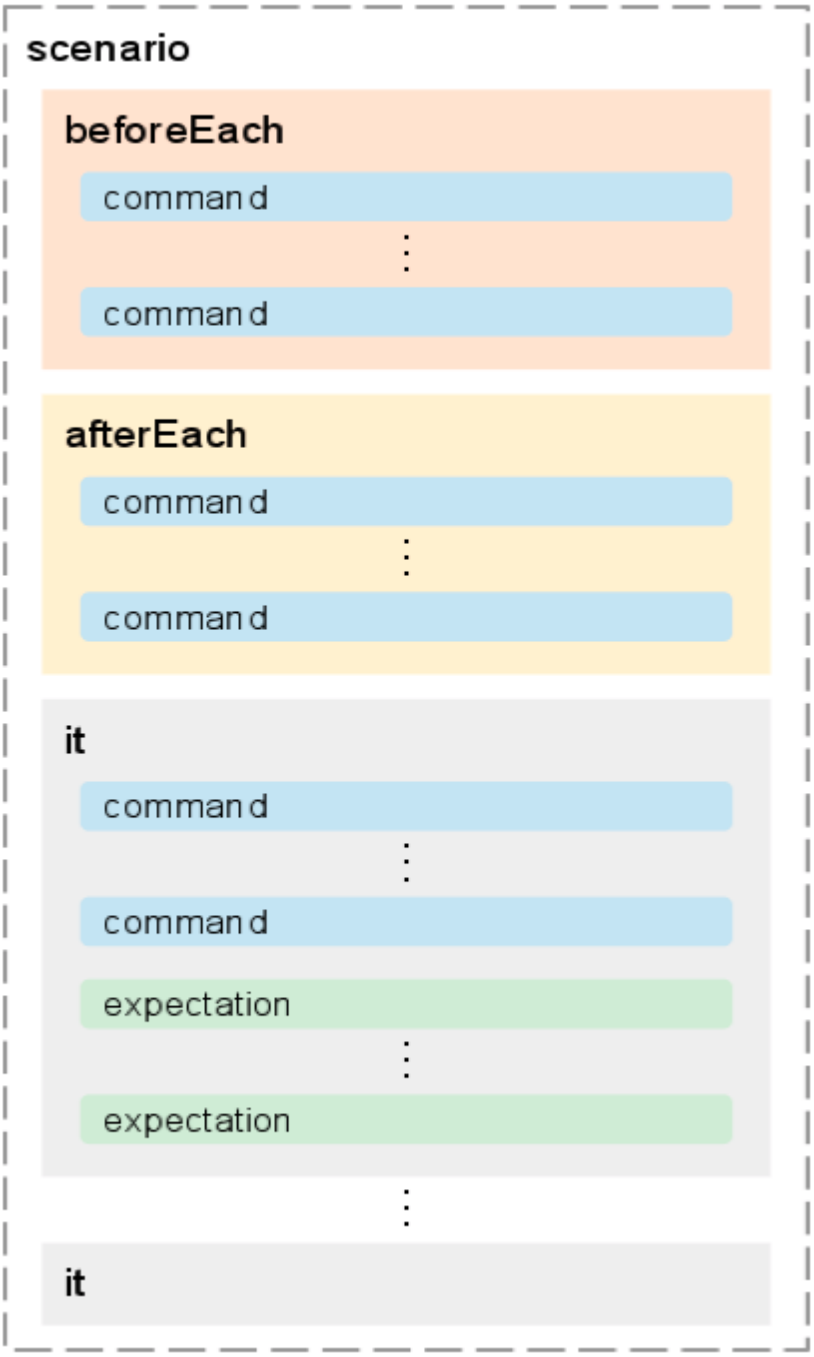
Protractor is a Node.js program, and runs end-to-end tests that are also written in JavaScript and run with node. Protractor uses WebDriverto control browsers and simulate user actions.

For more information on Protractor, view getting started or the api docs.

Protractor uses Jasmine for its test syntax. As in unit testing, a test file is comprised of one or more `it` blocks that describe the requirements of your application. `it` blocks are made of **commands** and **expectations**. Commands tell Protractor to do something with the application such as navigate to a page or click on a button. Expectations tell Protractor to assert something about the application's state, such as the value of a field or the current URL.

If any expectation within an `it` block fails, the runner marks the `it` as "failed" and continues on to the next block.

Test files may also have `beforeEach` and `afterEach` blocks, which will be run before or after each `it` block regardless of whether the block passes or fails.



In addition to the above elements, tests may also contain helper functions to avoid duplicating code in the `it` blocks.

Here is an example of a simple test:

```
describe('TODO list', function() {
  it('should filter results', function() {
```

```
    // Find the element with ng-model="user" and type "jacksparrow" into it
    element(by.model('user')).sendKeys('jacksparrow');

    // Find the first (and only) button on the page and click it
    element(by.css(':button')).click();

    // Verify that there are 10 tasks
    expect(element.all(by.repeater('task in tasks')).count()).toEqual(10);

    // Enter 'groceries' into the element with ng-model="filterText"
    element(by.model('filterText')).sendKeys('groceries');

    // Verify that now there is only one item in the task list
    expect(element.all(by.repeater('task in tasks')).count()).toEqual(1);
  });
});
```

This test describes the requirements of a ToDo list, specifically, that it should be able to filter the list of items.

Section 23

Using $location

The `$location` service parses the URL in the browser address bar (based on `window.location`) and makes the URL available to your application. Changes to the URL in the address bar are reflected into the `$location` service and changes to `$location` are reflected into the browser address bar.

The $location service:

- Exposes the current URL in the browser address bar, so you can
  - Watch and observe the URL.
  - Change the URL.
- Maintains synchronization between itself and the browser's URL when the user
  - Changes the address in the browser's address bar.

    - Clicks the back or forward button in the browser (or clicks a History link).
    - Clicks on a link in the page.
- Represents the URL object as a set of methods (protocol, host, port, path, search, hash).

Comparing `$location` to `window.location`

	window.location	**$location service**
purpose	allow read/write access to the current browser location	same
API	exposes "raw" object with properties that can be directly modified	exposes jQuery-st
integration with AngularJS application life-cycle	none	knows about all in integrates with $w
seamless integration with HTML5 API	no	yes (with a fallbacl
aware of docroot/context from which the application is loaded	no - window.location.pathname returns "/docroot/actual/path"	yes - $location.pat

---

When should I use `$location`?

Any time your application needs to react to a change in the current URL or if you want to change the current URL in the browser.

What does it not do?

It does not cause a full page reload when the browser URL is changed. To reload the page after changing the URL, use the lower-level API, `$window.location.href`.

General overview of the API

The `$location` service can behave differently, depending on the configuration that was provided to it when it was instantiated. The default configuration is suitable for many applications, for others customizing the configuration can enable new features.

Once the `$location` service is instantiated, you can interact with it via jQuery-style getter and setter methods that allow you to get or change the current URL in the browser.

`$location` service configuration

To configure the `$location` service, retrieve the $locationProvider and set the parameters as follows:

- **html5Mode(mode)**: `{boolean|Object}`
  `false` or `{enabled: false}` (default) - see Hashbang mode
  `true` or `{enabled: true}` - see HTML5 mode
  `{..., requireBase: true/false}` (only affects HTML5 mode) - see Relative links
  `{..., rewriteLinks: true/false/'string'}` (only affects HTML5 mode) - see HTML link rewriting
  Default:

```
{
  enabled: false,
  requireBase: true,
  rewriteLinks: true
}
```

- **hashPrefix(prefix)**: `{string}`
  Prefix used for Hashbang URLs (used in Hashbang mode or in legacy browsers in HTML5 mode).
  Default: `'!'`

Example configuration

```
$locationProvider.html5Mode(true).hashPrefix('*');
```

Getter and setter methods

`$location` service provides getter methods for read-only parts of the URL (absUrl, protocol, host, port) and getter / setter methods for url, path, search, hash:

```
// get the current path
$location.path();


// change the path
$location.path('/newValue')
```

All of the setter methods return the same `$location` object to allow chaining. For example, to change multiple segments in one go, chain setters like this:

```
$location.path('/newValue').search({key: value});
```

Replace method

There is a special `replace` method which can be used to tell the $location service that the next time the $location service is synced with the browser, the last history record should be replaced instead of creating a new one. This is useful when you want to implement redirection, which would otherwise break the back button (navigating back would retrigger the redirection). To change the current URL without creating a new browser history record you can call:

```
$location.path('/someNewPath');
$location.replace();
// or you can chain these as: $location.path('/someNewPath').replace();
```

Note that the setters don't update `window.location` immediately. Instead, the `$location` service is aware of the scope life-cycle and coalesces multiple `$location` mutations into one "commit" to the `window.location` object during the scope `$digest` phase. Since multiple changes to the $location's state will be pushed to the browser as a single change, it's enough to call the `replace()` method just once to make the entire "commit" a replace operation rather than an addition to the browser history. Once the browser is updated, the $location service resets the flag set by `replace()` method and future mutations will create new history records, unless `replace()` is called again.

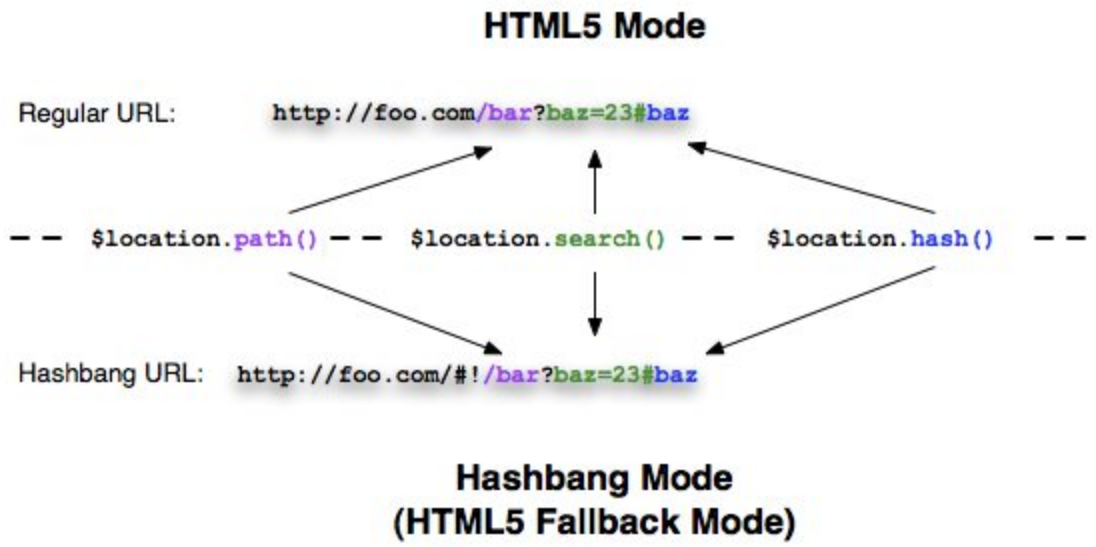
Setters and character encoding

You can pass special characters to `$location` service and it will encode them according to rules specified in RFC 3986. When you access the methods:

- All values that are passed to `$location` setter methods, `path()`, `search()`, `hash()`, are encoded.
- Getters (calls to methods without parameters) return decoded values for the following methods `path()`, `search()`, `hash()`.
- When you call the `absUrl()` method, the returned value is a full url with its segments encoded.
- When you call the `url()` method, the returned value is path, search and hash, in the form `/path?search=a&b=c#hash`. The segments are encoded as well.

Hashbang and HTML5 Modes

`$location` service has two configuration modes which control the format of the URL in the browser address bar: **Hashbang mode** (the default) and the **HTML5 mode** which is based on using the HTML5 History API. Applications use the same API in both modes and the `$location` service will work with appropriate URL segments and browser APIs to facilitate the browser URL change and history management.



	Hashbang mode	HTML5 mode
configuration	the default	{ html5Mode: true }

	Hashbang mode	HTML5 mode
URL format	hashbang URLs in all browsers	regular URLs in modern browser, hashba
<a href=""> link rewriting	no	yes
requires server-side configuration	no	yes

Hashbang mode (default mode)

In this mode, `$location` uses Hashbang URLs in all browsers. AngularJS also does not intercept and rewrite links in this mode. I.e. links work as expected and also perform full page reloads when other parts of the url than the hash fragment was changed.

Example

```
it('should show example', function() {
  module(function($locationProvider) {
    $locationProvider.html5Mode(false);
    $locationProvider.hashPrefix('!');
  });
  inject(function($location) {
    // open http://example.com/base/index.html#!/a

expect($location.absUrl()).toBe('http://example.com/base/index.html#!/a');
    expect($location.path()).toBe('/a');


    $location.path('/foo');

expect($location.absUrl()).toBe('http://example.com/base/index.html#!/foo');


    expect($location.search()).toEqual({});
    $location.search({a: 'b', c: true});
```

```
expect($location.absUrl()).toBe('http://example.com/base/index.html#!/foo?a=b
&c');


    $location.path('/new').search('x=y');

expect($location.absUrl()).toBe('http://example.com/base/index.html#!/new?x=y
');
  });
});
```

HTML5 mode

In HTML5 mode, the `$location` service getters and setters interact with the browser URL address through the HTML5 history API. This allows for use of regular URL path and search segments, instead of their hashbang equivalents. If the HTML5 History API is not supported by a browser, the `$location` service will fall back to using the hashbang URLs automatically. This frees you from having to worry about whether the browser displaying your app supports the history API or not; the `$location` service transparently uses the best available option.

- Opening a regular URL in a legacy browser -> redirects to a hashbang URL
- Opening hashbang URL in a modern browser -> rewrites to a regular URL

Note that in this mode, AngularJS intercepts all links (subject to the "Html link rewriting" rules below) and updates the url in a way that never performs a full page reload.

Example

```
it('should show example', function() {
  module(function($locationProvider) {
    $locationProvider.html5Mode(true);
    $locationProvider.hashPrefix('!');
  });
  inject(function($location) {
    // in browser with HTML5 history support:
    // open http://example.com/#!/a -> rewrite to http://example.com/a
    // (replacing the http://example.com/#!/a history record)
    expect($location.path()).toBe('/a');
```

```
    $location.path('/foo');
    expect($location.absUrl()).toBe('http://example.com/foo');

    expect($location.search()).toEqual({});
    $location.search({a: 'b', c: true});
    expect($location.absUrl()).toBe('http://example.com/foo?a=b&c');

    $location.path('/new').search('x=y');
    expect($location.url()).toBe('/new?x=y');
    expect($location.absUrl()).toBe('http://example.com/new?x=y');
  });
});


it('should show example (when browser doesn\'t support HTML5 mode',
function() {
  module(function($provide, $locationProvider) {
    $locationProvider.html5Mode(true);
    $locationProvider.hashPrefix('!');
    $provide.value('$sniffer', {history: false});
  });
  inject(initBrowser({ url: 'http://example.com/new?x=y', basePath: '/' }),
    function($location) {
    // in browser without html5 history support:
    // open http://example.com/new?x=y -> redirect to
http://example.com/#!/new?x=y
    // (again replacing the http://example.com/new?x=y history item)
    expect($location.path()).toBe('/new');
    expect($location.search()).toEqual({x: 'y'});

    $location.path('/foo/bar');
    expect($location.path()).toBe('/foo/bar');
    expect($location.url()).toBe('/foo/bar?x=y');
    expect($location.absUrl()).toBe('http://example.com/#!/foo/bar?x=y');
```

```
  });
});
```

Fallback for legacy browsers

For browsers that support the HTML5 history API, `$location` uses the HTML5 history API to write path and search. If the history API is not supported by a browser, `$location` supplies a Hashbang URL. This frees you from having to worry about whether the browser viewing your app supports the history API or not; the `$location` service makes this transparent to you.

HTML link rewriting

When you use HTML5 history API mode, you will not need special hashbang links. All you have to do is specify regular URL links, such as: `<a href="/some?foo=bar">link</a>`

When a user clicks on this link,

- In a legacy browser, the URL changes to `/index.html#!/some?foo=bar`
- In a modern browser, the URL changes to `/some?foo=bar`

In cases like the following, links are not rewritten; instead, the browser will perform a full page reload to the original link.

- Links that contain `target` element
  Example: `<a href="/ext/link?a=b" target="_self">link</a>`
- Absolute links that go to a different domain
  Example: `<a href="http://angularjs.org/">link</a>`
- Links starting with '/' that lead to a different base path
  Example: `<a href="/not-my-base/link">link</a>`

If `mode.rewriteLinks` is set to `false` in the `mode` configuration object passed to `$locationProvider.html5Mode()`, the browser will perform a full page reload for every link. `mode.rewriteLinks` can also be set to a string, which will enable link rewriting only on anchor elements that have the given attribute.

For example, if `mode.rewriteLinks` is set to `'internal-link'`:

- `<a href="/some/path" internal-link>link</a>` will be rewritten
- `<a href="/some/path">link</a>` will perform a full page reload

Note that attribute name normalization does not apply here,
so `'internalLink'` will **not** match `'internal-link'`.

Relative links

Be sure to check all relative links, images, scripts etc. AngularJS requires you to specify the url base in the head of your main html file (`<base href="/my-base/index.html">`)
unless `html5Mode.requireBase` is set to `false` in the html5Mode definition object passed

to `$locationProvider.html5Mode()`. With that, relative urls will always be resolved to this base url, even if the initial url of the document was different.

There is one exception: Links that only contain a hash fragment (e.g. `<a href="#target">`) will only change `$location.hash()` and not modify the url otherwise. This is useful for scrolling to anchors on the same page without needing to know on which page the user currently is.

Server side

Using this mode requires URL rewriting on server side, basically you have to rewrite all your links to entry point of your application (e.g. index.html). Requiring a `<base>` tag is also important for this case, as it allows AngularJS to differentiate between the part of the url that is the application base and the path that should be handled by the application.

Base href constraints

The `$location` service is not able to function properly if the current URL is outside the URL given as the base href. This can have subtle confusing consequences...

Consider a base href set as follows: `<base href="/base/">` (i.e. the application exists in the "folder" called `/base`). The URL `/base` is actually outside the application (it refers to the `base` file found in the root `/` folder).

If you wish to be able to navigate to the application via a URL such as `/base` then you should ensure that your server is setup to redirect such requests to `/base/`.

See https://github.com/angular/angular.js/issues/14018 for more information.

Sending links among different browsers

Because of rewriting capability in HTML5 mode, your users will be able to open regular url links in legacy browsers and hashbang links in modern browser:

- Modern browser will rewrite hashbang URLs to regular URLs.
- Older browsers will redirect regular URLs to hashbang URLs.

Example

Here you can see two `$location` instances that show the difference between **Html5 mode** and **Html5 Fallback mode**. Note that to simulate different levels of browser support, the `$location` instances are connected to a fakeBrowser service, which you don't have to set up in actual projects.

Note that when you type hashbang url into the first browser (or vice versa) it doesn't rewrite / redirect to regular / hashbang url, as this conversion happens only during parsing the initial URL = on page reload.

In these examples we use `<base href="/base/index.html" />`. The inputs represent the address bar of the browser.

Browser in HTML5 mode

Code: test_54.html, test_54.js

Browser in HTML5 Fallback mode (Hashbang mode)

Code: test_55.html, test_55.js

Caveats

Page reload navigation

The `$location` service allows you to change only the URL; it does not allow you to reload the page. When you need to change the URL and reload the page or navigate to a different page, please use a lower level API, $window.location.href.

Using $location outside of the scope life-cycle

`$location` knows about AngularJS's scope life-cycle. When a URL changes in the browser it updates the `$location` and calls `$apply` so that all $watchers / $observers are notified. When you change the `$location` inside the `$digest` phase everything is ok; `$location` will propagate this change into browser and will notify all the $watchers / $observers. When you want to change the `$location` from outside AngularJS (for example, through a DOM Event or during testing) - you must call `$apply` to propagate the changes.

$location.path() and ! or / prefixes

A path should always begin with forward slash (/); the `$location.path()` setter will add the forward slash if it is missing.

Note that the `!` prefix in the hashbang mode is not part of `$location.path()`; it is actually `hashPrefix`.

Crawling your app

To allow indexing of your AJAX application, you have to add special meta tag in the head section of your document:

```
<meta name="fragment" content="!" />
```

This will cause crawler bot to request links with `_escaped_fragment_` param so that your server can recognize the crawler and serve a HTML snapshots. For more information about this technique, see Making AJAX Applications Crawlable.

Testing with the $location service

When using `$location` service during testing, you are outside of the angular's scope life-cycle. This means it's your responsibility to call `scope.$apply()`.

```
describe('serviceUnderTest', function() {
  beforeEach(module(function($provide) {
    $provide.factory('serviceUnderTest', function($location) {
      // whatever it does...
    });
  });

  it('should...', inject(function($location, $rootScope, serviceUnderTest) {
    $location.path('/new/path');
    $rootScope.$apply();

    // test whatever the service should do...

  }));
});
```

Migrating from earlier AngularJS releases

In earlier releases of AngularJS, `$location` used `hashPath` or `hashSearch` to process path and search methods. With this release, the `$location` service processes path and search methods and then uses the information it obtains to compose hashbang URLs (such as`http://server.com/#!/path?search=a`), when necessary.

Changes to your code

Navigation inside the app	Change to
$location.href = value $location.hash = value $location.update(value) $location.updateHash(value)	$location.path(path).search(search)
$location.hashPath = path	$location.path(path)
$location.hashSearch = search	$location.search(search)

Navigation outside the app	Use lower level API
$location.href = value $location.update(value)	$window.location.href = value
$location[protocol | host | port | path | search]	$window.location[protocol | host | port | path | s

Read access	Change to
$location.hashPath	$location.path()
$location.hashSearch	$location.search()
$location.href $location.protocol $location.host $location.port	$location.absUrl() $location.protocol() $location.host() $location.port()

Navigation inside the app	Change to
$location.hash	$location.path() + $location.search()
$location.path $location.search	$window.location.path $window.location.search

Two-way binding to $location

Because `$location` uses getters/setters, you can use `ng-model-options="{ getterSetter: true }"` to bind it to `ngModel`:

Code: test_56.html, test_56.js

Section 24

Working with CSS

- `ng-scope`
  - **Usage:** AngularJS applies this class to any element for which a new scope is defined. (see scope guide for more information about scopes)
- `ng-isolate-scope`
  - **Usage:** AngularJS applies this class to any element for which a new isolate scope is defined.
- `ng-binding`
  - **Usage:** AngularJS applies this class to any element that is attached to a data binding, via `ng-bind` or `{{}}` curly braces, for example. (see databinding guide)
- `ng-invalid`, `ng-valid`
  - **Usage:** AngularJS applies this class to a form control widget element if that element's input does not pass validation. (see input directive)
- `ng-pristine`, `ng-dirty`
  - **Usage:** AngularJS ngModel directive applies `ng-pristine` class to a new form control widget which did not have user interaction. Once the user interacts with the form control, the class is changed to `ng-dirty`.

- `ng-touched`, `ng-untouched`
    - **Usage:** AngularJS ngModel directive applies `ng-untouched` class to a new form control widget which has not been blurred. Once the user blurs the form control, the class is changed to `ng-touched`.

Section 25

i18n and I10n

Internationalization (i18n) is the process of developing products in such a way that they can be localized for languages and cultures easily. Localization (l10n), is the process of adapting applications and text to enable their usability in a particular cultural or linguistic market. For application developers, internationalizing an application means abstracting all of the strings and other locale-specific bits (such as date or currency formats) out of the application. Localizing an application means providing translations and localized formats for the abstracted bits.

How does AngularJS support i18n/l10n?

AngularJS supports i18n/l10n for date, number and currency filters.

Localizable pluralization is supported via the `ngPluralize` directive. Additionally, you can use MessageFormat extensions to`$interpolate` for localizable pluralization and gender support in all interpolations via the `ngMessageFormat` module.

All localizable AngularJS components depend on locale-specific rule sets managed by the `$locale` service.

There are a few examples that showcase how to use AngularJS filters with various locale rule sets in the `i18n/e2e` directory of the AngularJS source code.

What is a locale ID?

A locale is a specific geographical, political, or cultural region. The most commonly used locale ID consists of two parts: language code and country code. For example, `en-US`, `en-AU`, and `zh-CN` are all valid locale IDs that have both language codes and country codes. Because specifying a country code in locale ID is optional, locale IDs such as `en`, `zh`, and `sk` are also valid. See the ICU website for more information about using locale IDs.

Supported locales in AngularJS

AngularJS separates number and datetime format rule sets into different files, each file for a particular locale. You can find a list of currently supported locales here

Providing locale rules to AngularJS

There are two approaches to providing locale rules to AngularJS:

1. Pre-bundled rule sets

You can pre-bundle the desired locale file with AngularJS by concatenating the content of the locale-specific file to the end of `angular.js` or `angular.min.js` file.

For example on *nix, to create an angular.js file that contains localization rules for german locale, you can do the following:

```
cat angular.js i18n/angular-locale_de-de.js > angular_de-de.js
```

When the application containing `angular_de-de.js` script instead of the generic angular.js script starts, AngularJS is automatically pre-configured with localization rules for the german locale.

2. Including a locale script in `index.html`

You can also include the locale specific js file in the index.html page. For example, if one client requires German locale, you would serve index_de-de.html which will look something like this:

```
<html ng-app>
 <head>
….
   <script src="angular.js"></script>
   <script src="i18n/angular-locale_de-de.js"></script>
….
 </head>
</html>
```

Comparison of the two approaches

Both approaches described above require you to prepare different `index.html` pages or JavaScript files for each locale that your app may use. You also need to configure your server to serve the correct file that corresponds to the desired locale.

The second approach (including the locale JavaScript file in `index.html`) may be slower because an extra script needs to be loaded.

Caveats

Although AngularJS makes i18n convenient, there are several things you need to be conscious of as you develop your app.

Currency symbol

AngularJS's currency filter allows you to use the default currency symbol from the locale service, or you can provide the filter with a custom currency symbol.

Best Practice: If your app will be used only in one locale, it is fine to rely on the default currency symbol. If you anticipate that viewers in other locales might use your app, you should explicitly provide a currency symbol.

Let's say you are writing a banking app and you want to display an account balance of 1000 dollars. You write the following binding using the currency filter:

```
{{ 1000 | currency }}
```

If your app is currently in the `en-US` locale, the browser will show `$1000.00`. If someone in the Japanese locale (`ja`) views your app, their browser will show a balance of `¥1000.00` instead. This is problematic because $1000 is not the same as ¥1000.

In this case, you need to override the default currency symbol by providing the `currency` currency filter with a currency symbol as a parameter.

If we change the above to `{{ 1000 | currency:"USD$"}}`, AngularJS will always show a balance of `USD$1000` regardless of locale.

Translation length

Translated strings/datetime formats can vary greatly in length. For example, `June 3, 1977` will be translated to Spanish as `3 de junio de 1977`.

When internationalizing your app, you need to do thorough testing to make sure UI components behave as expected even when their contents vary greatly in content size.

Timezones

The AngularJS datetime filter uses the time zone settings of the browser. The same application will show different time information depending on the time zone settings of the computer that the application is running on. Neither JavaScript nor AngularJS currently supports displaying the date with a timezone specified by the developer.

MessageFormat extensions

You can write localizable plural and gender based messages in AngularJS interpolation expressions and `$interpolate` calls.

This syntax extension is provided by way of the `ngMessageFormat` module that your application can depend upon (shipped separately as `angular-message-format.min.js` and `angular-message-format.js`.) A current limitation of the `ngMessageFormat` module, is that it does not support redefining the `$interpolate` start and end symbols. Only the default `{{` and `}}` are allowed.

The syntax extension is based on a subset of the ICU MessageFormat syntax that covers plurals and gender selections. Please refer to the links in the “Further Reading” section at the bottom of this section.

You may find it helpful to play with the following example as you read the explanations below:

Code: test_57.html, test_57.js

Plural Syntax

The syntax for plural based message selection looks like the following:

```
{{NUMERIC_EXPRESSION, plural,
    =0 {MESSAGE_WHEN_VALUE_IS_0}
    =1 {MESSAGE_WHEN_VALUE_IS_1}
    =2 {MESSAGE_WHEN_VALUE_IS_2}
    =3 {MESSAGE_WHEN_VALUE_IS_3}
    ...
    zero {MESSAGE_WHEN_PLURAL_CATEGORY_IS_ZERO}
    one {MESSAGE_WHEN_PLURAL_CATEGORY_IS_ONE}
    two {MESSAGE_WHEN_PLURAL_CATEGORY_IS_TWO}
    few {MESSAGE_WHEN_PLURAL_CATEGORY_IS_FEW}
    many {MESSAGE_WHEN_PLURAL_CATEGORY_IS_MANY}
    other {MESSAGE_WHEN_THERE_IS_NO_MATCH}
}}
```

Please note that whitespace (including newline) is generally insignificant except as part of the actual message text that occurs in curly braces. Whitespace is generally used to aid readability.

Here, `NUMERIC_EXPRESSION` is an expression that evaluates to a numeric value based on which the displayed message should change based on pluralization rules.

Following the AngularJS expression, you would denote the plural extension syntax by the `, plural,` syntax element. The spaces there are optional.

This is followed by a list of selection keyword and corresponding message pairs. The "other" keyword and corresponding message are **required** but you may have as few or as many of the other categories as you need.

Selection Keywords

The selection keywords can be either exact matches or language dependent plural categories.

Exact matches are written as the equal sign followed by the exact value. `=0`, `=1`, `=2` and `=123` are all examples of exact matches. Note that there should be no space between the equal sign and the numeric value.

Plural category matches are single words corresponding to the plural categories of the CLDR plural category spec. These categories vary by locale. The "en" (English) locale, for example, defines just "one" and "other" while the "ga" (Irish) locale defines "one", "two", "few", "many" and "other". Typically, you would just write the categories for your language. During translation, the translators will add or remove more categories depending on the target locale.

Exact matches always win over keyword matches. Therefore, if you define both `=0` and `zero`, when the value of the expression is zero, the `=0` message is the one that will be selected. (The duplicate keyword categories are helpful when used with the optional `offset`syntax described later.)

Messages

Messages immediately follow a selection keyword and are optionally preceded by whitespace. They are written in single curly braces (`{}`). They may contain AngularJS interpolation syntax inside them. In addition, the `#` symbol is a placeholder for the actual numeric value of the expression.

Simple plural example

```
{{numMessages, plural,
      =0 {You have no new messages}
      =1 {You have one new message}
   other {You have # new messages}
}}
```

Because these messages can themselves contain AngularJS expressions, you could also write this as follows:

```
{{numMessages, plural,
      =0 {You have no new messages}
      =1 {You have one new message}
```

```
   other {You have {{numMessages}} new messages}
}}
```

Plural syntax with optional `offset`

The plural syntax supports an optional `offset` syntax that is used in matching. It's simpler to explain this with an example.

```
{{recipients.length, plural, offset:1
    =0    {You gave no gifts}
    =1    {You gave {{recipients[0].name}} a gift}
    one   {You gave {{recipients[0].name}} and one other person a gift}
    other {You gave {{recipients[0].name}} and # other people a gift}
}}
```

When an `offset` is specified, the matching works as follows. First, the exact value of the AngularJS expression is matched against the exact matches (i.e. `=N` selectors) to find a match. If there is one, that message is used. If there was no match, then the offset value is subtracted from the value of the expression and locale specific pluralization rules are applied to this new value to obtain its plural category (such as “one”, “few”, “many”, etc.) and a match is attempted against the keyword selectors and the matching message is used. If there was no match, then the “other” category (required) is used. The value of the `#` character inside a message is the value of original expression reduced by the offset value that was specified.

Escaping / Quoting

You will need to escape curly braces or the `#` character inside message texts if you want them to be treated literally with no special meaning. You may quote/escape any character in your message text by preceding it with a `\` (backslash) character. The backslash character removes any special meaning to the character that immediately follows it. Therefore, you can escape or quote the backslash itself by preceding it with another backslash character.

Gender (aka select) Syntax

The gender support is provided by the more generic "select" syntax that is more akin to a switch statement. It is general enough to support use for gender based messages.

The syntax for gender based message selection looks like the following:

```
{{EXPRESSION, select,
    male {MESSAGE_WHEN_EXPRESSION_IS_MALE}
    female {MESSAGE_WHEN_EXPRESSION_IS_FEMALE}
    ...
```

```
    other {MESSAGE_WHEN_THERE_IS_NO_GENDER_MATCH}
}}
```

Please note that whitespace (including newline) is generally insignificant except as part of the actual message text that occurs in curly braces. Whitespace is generally used to aid readability.

Here, `EXPRESSION` is an AngularJS expression that evaluates to the gender of the person that is used to select the message that should be displayed.

The AngularJS expression is followed by `, select,` where the spaces are optional.

This is followed by a list of selection keyword and corresponding message pairs. The "other" keyword and corresponding message are **required** but you may have as few or as many of the other gender values as you need (i.e. it isn't restricted to male/female.) Note however, that the matching is **case-sensitive**.

Selection Keywords

Selection keywords are simple words like "male" and "female". The keyword, "other", and its corresponding message are required while others are optional. It is used when the AngularJS expression does not match (case-insensitively) any of the other keywords specified.

Messages

Messages immediately follow a selection keyword and are optionally preceded by whitespace. They are written in single curly braces (`{}`). They may contain AngularJS interpolation syntax inside them.

Simple gender example

```
{{friendGender, select,
       male {Invite him}
     female {Invite her}
      other {Invite them}
}}
```

Nesting

As mentioned in the syntax for plural and select, the embedded messages can contain AngularJS interpolation syntax. Since you can use MessageFormat extensions in AngularJS interpolation, this allows you to nest plural and gender expressions in any order.

Please note that if these are intended to reach a translator and be translated, it is recommended that the messages appear as a whole and not be split up.

Demonstration of nesting

This is taken from the above example.

```
{{recipients.length, plural, offset:1
    =0 {You ({{sender.name}}) gave no gifts}
    =1 { {{ recipients[0].gender, select,
               male {You ({{sender.name}}) gave him ({{recipients[0].name}}) a
gift.}
               female {You ({{sender.name}}) gave her ({{recipients[0].name}})
a gift.}
               other {You ({{sender.name}}) gave them ({{recipients[0].name}})
a gift.}
          }}
       }
    one { {{ recipients[0].gender, select,
               male {You ({{sender.name}}) gave him ({{recipients[0].name}})
and one other person a gift.}
               female {You ({{sender.name}}) gave her ({{recipients[0].name}})
and one other person a gift.}
               other {You ({{sender.name}}) gave them ({{recipients[0].name}})
and one other person a gift.}
          }}
       }
    other {You ({{sender.name}}) gave {{recipients.length}} people gifts. }
}}
```

Differences from the ICU MessageFormat syntax

This section is useful to you if you're already familiar with the ICU MessageFormat syntax.

This syntax extension, while based on MessageFormat, has been designed to be backwards compatible with existing AngularJS interpolation expressions. The key rule is simply this: **All interpolations are done inside double curlies.** The top level comma operator after an expression inside the double curlies causes MessageFormat extensions to be recognized. Such a top level comma is otherwise illegal in an AngularJS expression and is used by MessageFormat to specify the function (such as plural/select) and it's related syntax.

To understand the extension, take a look at the ICU MessageFormat syntax as specified by the ICU documentation. Anywhere in that MessageFormat that you have regular message text and you want to substitute an expression, just put it in double curlies instead of single curlies that MessageFormat dictates. This has a huge advantage. **You are no longer limited to simple**

identifiers for substitutions. Because you are using double curlies, you can stick in any arbitrary interpolation syntax there, including nesting more MessageFormat expressions!

Section 26

Security

Use the latest AngularJS possible

Like any software library, it is critical to keep AngularJS up to date. Please track the CHANGELOG and make sure you are aware of upcoming security patches and other updates.

Be ready to update rapidly when new security-centric patches are available.

Those that stray from AngularJS standards (such as modifying AngularJS's core) may have difficulty updating, so keeping to AngularJS standards is not just a functionality issue, it's also critical in order to facilitate rapid security updates.

AngularJS Templates and Expressions

If an attacker has access to control AngularJS templates or expressions, they can exploit an AngularJS application via an XSS attack, regardless of the version.

There are a number of ways that templates and expressions can be controlled:

- **Generating AngularJS templates on the server containing user-provided content**. This is the most common pitfall where you are generating HTML via some server-side engine such as PHP, Java or ASP.NET.
- **Passing an expression generated from user-provided content in calls to the following methods on a scope**:
    - `$watch(userContent, ...)`
    - `$watchGroup(userContent, ...)`
    - `$watchCollection(userContent, ...)`
    - `$eval(userContent)`
    - `$evalAsync(userContent)`
    - `$apply(userContent)`
    - `$applyAsync(userContent)`
- **Passing an expression generated from user-provided content in calls to services that parse expressions**:

    - `$compile(userContent)`
    - `$parse(userContent)`
    - `$interpolate(userContent)`
- **Passing an expression generated from user provided content as a predicate to `orderBy` pipe**:`{{ value | orderBy : userContent }}`

Sandbox removal

Each version of AngularJS 1 up to, but not including 1.6, contained an expression sandbox, which reduced the surface area of the vulnerability but never removed it. **In AngularJS 1.6 we removed this sandbox as developers kept relying upon it as a security feature even though it was always possible to access arbitrary JavaScript code if one could control the AngularJS templates or expressions of applications.**

Control of the AngularJS templates makes applications vulnerable even if there was a completely secure sandbox:

- https://ryhanson.com/stealing-session-tokens-on-plunker-with-an-angular-expression-injection/ in this blog post the author shows a (now closed) vulnerability in the Plunker application due to server-side rendering inside an AngularJS template.
- https://ryhanson.com/angular-expression-injection-walkthrough/ in this blog post the author describes an attack, which does not rely upon an expression sandbox bypass, that can be made because the sample application is rendering a template on the server that contains user entered content.

It's best to design your application in such a way that users cannot change client-side templates.

- Do not mix client and server templates
- Do not use user input to generate templates dynamically
- Do not run user input through `$scope.$eval` (or any of the other expression parsing functions listed above)
- Consider using CSP (but don't rely only on CSP)

You can use suitably sanitized server-side templating to dynamically generate CSS, URLs, etc, but not for generating templates that are bootstrapped/compiled by AngularJS.

If you must continue to allow user-provided content in an AngularJS template then the safest option is to ensure that it is only present in the part of the template that is made inert via the `ngNonBindable` directive.

HTTP Requests

Whenever your application makes requests to a server there are potential security issues that need to be blocked. Both server and the client must cooperate in order to eliminate these threats. AngularJS comes pre-configured with strategies that address these issues, but for this to work backend server cooperation is required.

Cross Site Request Forgery (XSRF/CSRF)

Protection from XSRF is provided by using the double-submit cookie defense pattern. For more information please visit XSRF protection.

JSON Hijacking Protection

Protection from JSON Hijacking is provided if the server prefixes all JSON requests with following string `")]}',\n"`. AngularJS will automatically strip the prefix before processing it as JSON. For more information please visit JSON Hijacking Protection.

Bear in mind that calling `$http.jsonp` gives the remote server (and, if the request is not secured, any Man-in-the-Middle attackers) instant remote code execution in your application: the result of these requests is handed off to the browser as regular `<script>` tag.

Strict Contextual Escaping

Strict Contextual Escaping (SCE) is a mode in which AngularJS requires bindings in certain contexts to require a value that is marked as safe to use for that context.

This mode is implemented by the `$sce` service and various core directives.

One example of such a context is rendering arbitrary content via the `ngBindHtml` directive. If the content is provided by a user there is a chance of Cross Site Scripting (XSS) attacks.
The `ngBindHtml` directive will not render content that is not marked as safe by `$sce`.
The `ngSanitize` module can be used to clean such user provided content and mark the content as safe.

Be aware that marking untrusted data as safe via calls to `$sce.trustAsHtml`, etc is dangerous and will lead to Cross Site Scripting exploits.

For more information please visit `$sce` and `$sanitize`.

Using Local Caches

There are various places that the browser can store (or cache) data. Within AngularJS there are objects created by the `$cacheFactory`. These objects, such as `$templateCache` are used to store and retrieve data, primarily used by `$http` and the `script` directive to cache templates and other data.

Similarly the browser itself offers `localStorage` and `sessionStorage` objects for caching data.

Attackers with local access can retrieve sensitive data from this cache even when users are not authenticated.

For instance in a long running Single Page Application (SPA), one user may "log out", but then another user may access the application without refreshing, in which case all the cached data is still available.

For more information please visit Web Storage Security.

Section 27

Accessibility

The goal of ngAria is to improve AngularJS's default accessibility by enabling common ARIA attributes that convey state or semantic information for assistive technologies used by persons with disabilities.

Including ngAria

Using ngAria is as simple as requiring the ngAria module in your application. ngAria hooks into standard AngularJS directives and quietly injects accessibility support into your application at runtime.

```
angular.module('myApp', ['ngAria'])...
```

Using ngAria

Most of what ngAria does is only visible "under the hood". To see the module in action, once you've added it as a dependency, you can test a few things:

- Using your favorite element inspector, look for attributes added by ngAria in your own code.
- Test using your keyboard to ensure `tabindex` is used correctly.
- Fire up a screen reader such as VoiceOver or NVDA to check for ARIA support. Helpful screen reader tips.

Supported directives

Currently, ngAria interfaces with the following directives:

- ngModel

- ngDisabled
- ngRequired
- ngReadonly
- ngChecked
- ngValue
- ngShow
- ngHide
- ngClick
- ngDblClick
- ngMessages

---

ngModel

Much of ngAria's heavy lifting happens in the ngModel directive. For elements using ngModel, special attention is paid by ngAria if that element also has a role or type of `checkbox`, `radio`, `range` or `textbox`.

For those elements using ngModel, ngAria will dynamically bind and update the following ARIA attributes (if they have not been explicitly specified by the developer):

- aria-checked
- aria-valuemin
- aria-valuemax
- aria-valuenow
- aria-invalid
- aria-required
- aria-readonly
- aria-disabled

Example

Code: test_58.html, test_58.js

ngAria will also add `tabIndex`, ensuring custom elements with these roles will be reachable from the keyboard. It is still up to **you** as a developer to **ensure custom controls will be accessible**. As a rule, any time you create a widget involving user interaction, be sure to test it with your keyboard and at least one mobile and desktop screen reader.

ngValue and ngChecked

To ease the transition between native inputs and custom controls, ngAria now supports ngValue and ngChecked. The original directives were created for native inputs only, so ngAria extends support to custom elements by managing `aria-checked` for accessibility.

Example

```
<custom-checkbox ng-checked="val"></custom-checkbox>

<custom-radio-button ng-value="val"></custom-radio-button>
```

Becomes:

```
<custom-checkbox ng-checked="val" aria-checked="true"></custom-checkbox>

<custom-radio-button ng-value="val" aria-checked="true"></custom-radio-
button>
```

ngDisabled

The `disabled` attribute is only valid for certain elements such as `button`, `input` and `textarea`. To properly disable custom element directives such as `<md-checkbox>` or `<taco-tab>`, using ngAria with ngDisabled will also add `aria-disabled`. This tells assistive technologies when a non-native input is disabled, helping custom controls to be more accessible.

Example

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

Becomes:

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

You can check whether a control is legitimately disabled for a screen reader by visiting chrome://accessibility and inspecting the accessibility tree.

ngRequired

The boolean `required` attribute is only valid for native form controls such as `input` and `textarea`. To properly indicate custom element directives such as `<md-checkbox>` or `<custom-input>` as required, using ngAria with ngRequired will also add`aria-required`. This tells accessibility APIs when a custom control is required.

Example

```
<md-checkbox ng-required="val"></md-checkbox>
```

Becomes:

```
<md-checkbox ng-required="val" aria-required="true"></md-checkbox>
```

ngReadonly

The boolean `readonly` attribute is only valid for native form controls such as `input` and `textarea`. To properly indicate custom element directives such as `<md-checkbox>` or `<custom-input>` as required, using ngAria with ngReadonly will also add`aria-readonly`. This tells accessibility APIs when a custom control is read-only.

Example

```
<md-checkbox ng-readonly="val"></md-checkbox>
```

Becomes:

```
<md-checkbox ng-readonly="val" aria-readonly="true"></md-checkbox>
```

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.

In its default setup, ngAria for `ngShow` is actually redundant. It toggles `aria-hidden` on the directive when it is hidden or shown. However, the default CSS of `display: none !important`, already

hides child elements from a screen reader. It becomes more useful when the default CSS is overridden with properties that don't affect assistive technologies, such as `opacity` or `transform`. By toggling `aria-hidden` dynamically with ngAria, we can ensure content visually hidden with this technique will not be read aloud in a screen reader.

One caveat with this combination of CSS and `aria-hidden`: you must also remove links and other interactive child elements from the tab order using `tabIndex="-1"` on each control. This ensures screen reader users won't accidentally focus on "mystery elements". Managing tab index on every child control can be complex and affect performance, so it's best to just stick with the default `display: none` CSS. See the fourth rule of ARIA use.

Example

```
.ng-hide {
  display: block;
  opacity: 0;
}
<div ng-show="false" class="ng-hide" aria-hidden="true"></div>
```

Becomes:

```
<div ng-show="true" aria-hidden="false"></div>
```

Note: Child links, buttons or other interactive controls must also be removed from the tab order.

---

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 default CSS for `ngHide`, the inverse method to `ngShow`, makes ngAria redundant. It toggles `aria-hidden` on the directive when it is hidden or shown, but the content is already hidden with `display: none`. See explanation for ngShow when overriding the default CSS.

---

ngClick and ngDblclick

If `ng-click` or `ng-dblclick` is encountered, ngAria will add `tabindex="0"` to any element not in a node blacklist: *Button* Anchor *Input*Textarea *Select* Details/Summary To fix widespread accessibility problems with `ng-click` on `div` elements, ngAria will dynamically bind a keypress event by default as long as the element isn't in the node blacklist. You can turn this functionality on or off with the `bindKeypress` configuration option. ngAria will also add the `button` role to

communicate to users of assistive technologies. This can be disabled with the `bindRoleForClick` configuration option. For `ng-dblclick`, you must still manually add `ng-keypress` and a role to non-interactive elements such as `div` or `taco-button` to enable keyboard access.

Example

`html <div ng-click="toggleMenu()"></div>` Becomes: `html <div ng-click="toggleMenu()" tabindex="0"></div>`

---

ngMessages

The ngMessages module makes it easy to display form validation or other messages with priority sequencing and animation. To expose these visual messages to screen readers, ngAria injects `aria-live="assertive"`, causing them to be read aloud any time a message is shown, regardless of the user's focus location.

Example

```
<div ng-messages="myForm.myName.$error">
  <div ng-message="required">You did not enter a field</div>
  <div ng-message="maxlength">Your field is too long</div>
</div>
```

Becomes:

```
<div ng-messages="myForm.myName.$error" aria-live="assertive">
  <div ng-message="required">You did not enter a field</div>
  <div ng-message="maxlength">Your field is too long</div>
</div>
```

Disabling attributes

The attribute magic of ngAria may not work for every scenario. To disable individual attributes, you can use the config method. Just keep in mind this will tell ngAria to ignore the attribute globally.

Code: test_59.html, test_59.js

Common Accessibility Patterns

Accessibility best practices that apply to web apps in general also apply to AngularJS.

- **Text alternatives**: Add alternate text content to make visual information accessible using these W3C guidelines. The appropriate technique depends on the specific markup but can be accomplished using offscreen spans, `aria-label` or label elements, image `alt` attributes, `figure`/`figcaption` elements and more.
- **HTML Semantics**: If you're creating custom element directives, Web Components or HTML in general, use native elements wherever possible to utilize built-in events and properties. Alternatively, use ARIA to communicate semantic meaning. See notes on ARIA use.
- **Focus management**: Guide the user around the app as views are appended/removed. Focus should *never* be lost, as this causes unexpected behavior and much confusion (referred to as "freak-out mode").
- **Announcing changes**: When filtering or other UI messaging happens away from the user's focus, notify with ARIA Live Regions.
- **Color contrast and scale**: Make sure content is legible and interactive controls are usable at all screen sizes. Consider configurable UI themes for people with color blindness, low vision or other visual impairments.
- **Progressive enhancement**: Some users do not browse with JavaScript enabled or do not have the latest browser. An accessible message about site requirements can inform users and improve the experience.