Bootstrap

This page explains the Angular initialization process and how you can manually initialize Angular if necessary.

Angular `<script>` Tag

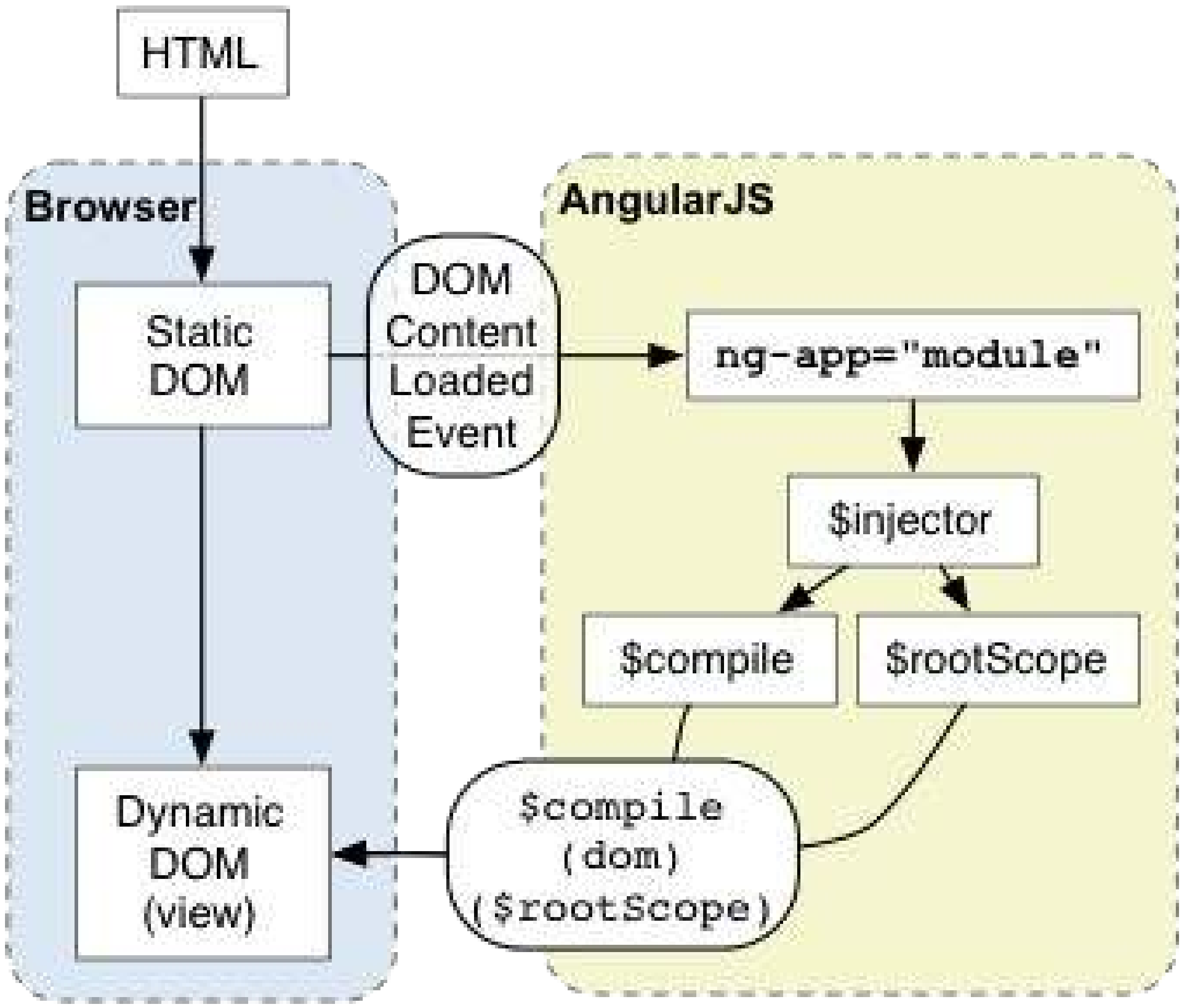
This example shows the recommended path for integrating Angular with what we call automatic initialization.

```
<!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 angular to auto-bootstrap your application.

3. If you choose to use the old style directive syntax `ng:` then include xml-namespace in `html` to make IE happy. (This is here for historical reasons, and we no longer recommend use of `ng:`.)

Automatic Initialization



Angular 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 Angular looks for the ngApp directive which designates your application root. If the ngApp directive is found then Angular 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 Angular 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 Angular compiles a page.

Here is an example of manually initializing Angular:

```
<!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 modules before 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 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.

Bootstrapping Angular Applications

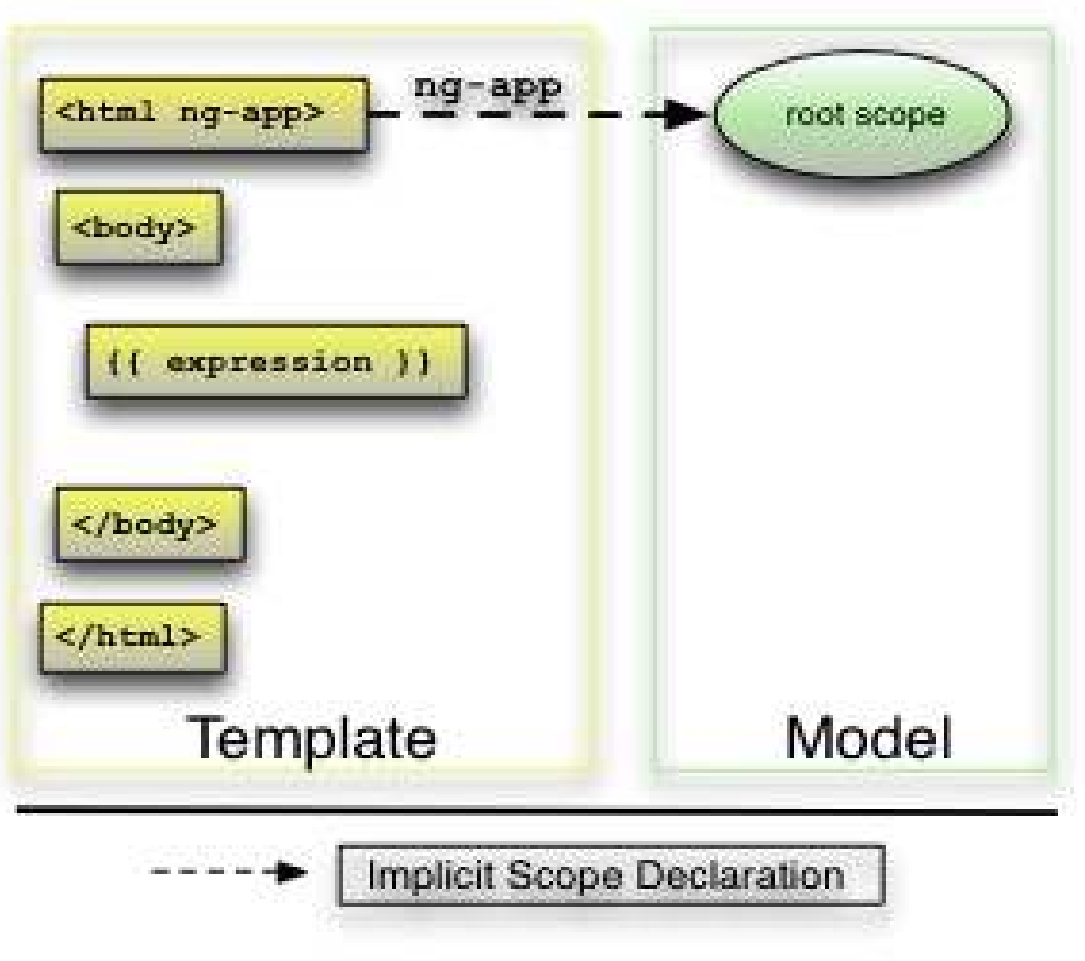
Bootstrapping Angular applications automatically using the `ngApp` directive is very easy and suitable for most cases. In advanced cases, such as when using script loaders, you can use the imperative/manual way to bootstrap the application.

There are 3 important things that happen during the bootstrap phase:

1. The injector that will be used for dependency injection is created.
2. The injector will then create the root scope that will become the context for the model of our application.
3. Angular will then "compile" the DOM starting at the `ngApp` root element, processing any directives and bindings found along the way.

Once an application is bootstrapped, it will then wait for incoming browser events (such as mouse clicks, key presses or incoming HTTP responses) that might change the model. Once such an event occurs, Angular detects if it caused any model changes and if changes are found, Angular will reflect them in the view by updating all of the affected bindings.

The structure of our application is currently very simple. The template contains just one directive and one static binding, and our model is empty. That will soon change!



$injector

1. - service in module auto

$injector is used to retrieve object instances as defined by provider, instantiate types, invoke methods, and load modules.

The following always holds true:

```
var $injector = angular.injector();
expect($injector.get('$injector')).toBe($injector);
expect($injector.invoke(function($injector) {
  return $injector;
})).toBe($injector);
```

$rootScope

1. - $rootScopeProvider
2. - service in module ng

Every application has a single root scope. All other scopes are descendant scopes of the root scope. Scopes provide separation between the model and the view, via a mechanism for watching the model for changes. They also provide event emission/broadcast and subscription facility. See the developer guide on scopes.

$rootScope.Scope

1. - type in module ng

A root scope can be retrieved using the $rootScope key from the $injector. Child scopes are created using the $new() method. (Most scopes are created automatically when compiled HTML template is executed.) See also the Scopes guide for an in-depth introduction and usage examples.

Inheritance

A scope can inherit from a parent scope, as in this example:

```
var parent = $rootScope;
var child = parent.$new();
```

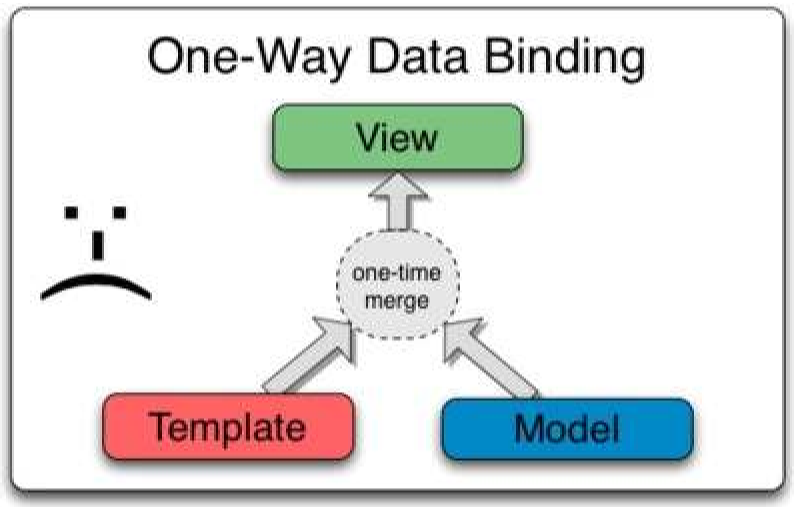
```
parent.salutation = "Hello";
expect(child.salutation).toEqual('Hello');


child.salutation = "Welcome";
expect(child.salutation).toEqual('Welcome');
expect(parent.salutation).toEqual('Hello');
```

When interacting with `Scope` in tests, additional helper methods are available on the instances of `Scope` type. See ngMock Scope for additional details.

Data Binding

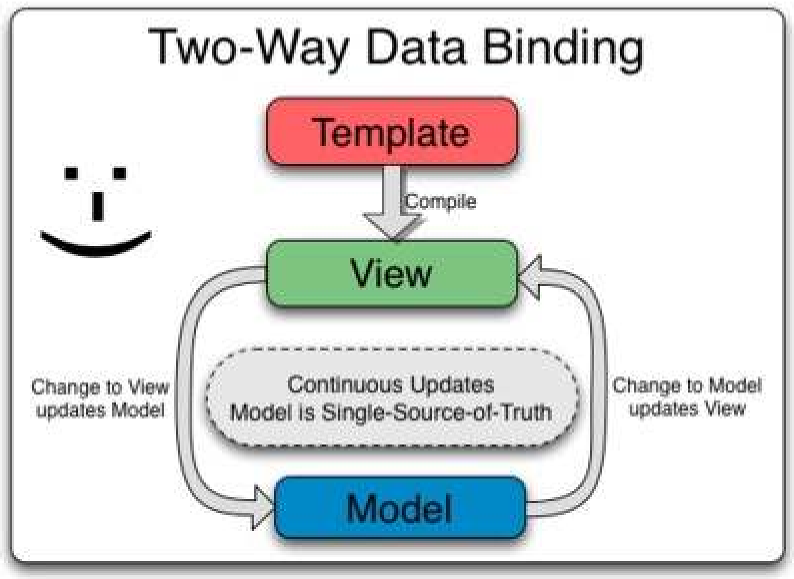
Data-binding in Angular apps is the automatic synchronization of data between the model and view components. The way that Angular implements data-binding lets you treat the model as the single-source-of-truth in your application. The view is a projection of the model at all times. When the model changes,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


Two-Way Data Binding
Template
Compile
View
Change to View updates Model
Continuous Updates Model is Single-Source-of-Truth
Change to Model updates View
Model

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.

Understanding Controllers

In Angular, a Controller is defined by 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 created and made available as an injectable parameter to the Controller's constructor function as `$scope`.

If the controller has been attached using the `controller as` syntax then the controller instance will be assigned to a property on the new 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 and directives 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!';
}]);
```

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`).

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 watch expressions 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 scopes for 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.

Dependency Injection

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

The Angular 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 Angular. 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

Angular 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:

```html
<!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) {
  // Angular will throw when this runs
}]);
```

When the `willBreak` service is instantiated, Angular 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 Angular'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.

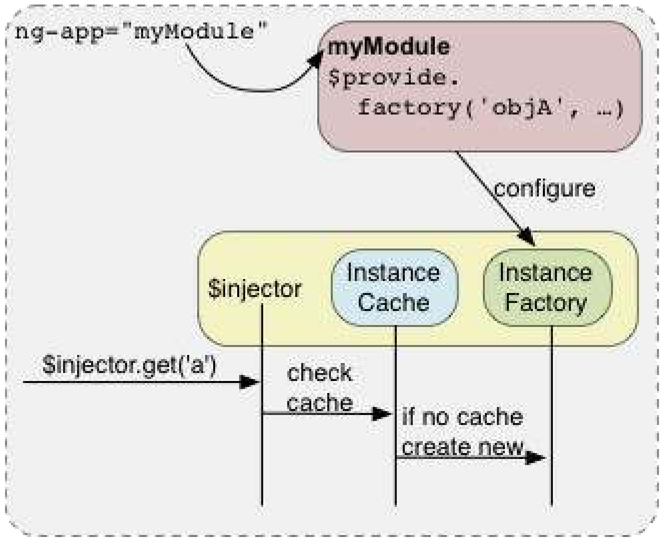
The first two options of creating or looking up dependencies are not optimal because they hard code the dependency to the component. This makes it difficult, if not impossible, to modify the dependencies. This is especially problematic in tests, where it is often desirable to provide mock dependencies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

Note: Angular uses **constructor injection**.

Interpolation and data-binding

Interpolation markup with embedded expressions is used by Angular 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 Angular 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, Angular 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>
```

Filters

Filters format the value of an expression for display to the user. They can be used in view templates, controllers or services. Angular 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 Angular 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.

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.

Edit in Plunker

index.html

```
<div ng-controller="ExampleController">
  <form novalidate class="simple-form">
    <label>Name: <input type="text" ng-model="user.name" /></label><br />
    <label>E-mail: <input type="email" ng-model="user.email" /></label><br />
    Best Editor: <label><input type="radio" ng-model="user.preference" value="vi" />vi</label>
    <label><input type="radio" ng-model="user.preference" value="emacs" />emacs</label><br />
    <input type="button" ng-click="reset()" value="Reset" />
    <input type="submit" ng-click="update(user)" value="Save" />
  </form>
  <pre>user = {{user | json}}</pre>
  <pre>master = {{master | json}}</pre>
</div>

<script>
  angular.module('formExample', [])
    .controller('ExampleController', ['$scope', function($scope) {
      $scope.master = {};

      $scope.update = function(user) {
        $scope.master = angular.copy(user);
      };

      $scope.reset = function() {
        $scope.user = angular.copy($scope.master);
      };


      $scope.reset();
    }]);
</script>
```

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.

Edit in Plunker

index.html

```
<div ng-controller="ExampleController">
  <form novalidate class="css-form">
    <label>Name: <input type="text" ng-model="user.name" required /></label><br />
    <label>E-mail: <input type="email" ng-model="user.email" required /></label><br />
    Gender: <label><input type="radio" ng-model="user.gender" value="male" />male</label>
    <label><input type="radio" ng-model="user.gender" value="female" />female</label><br />
    <input type="button" ng-click="reset()" value="Reset" />
    <input type="submit" ng-click="update(user)" value="Save" />
  </form>
  <pre>user = {{user | json}}</pre>
  <pre>master = {{master | json}}</pre>
</div>

<style type="text/css">
  .css-form input.ng-invalid.ng-touched {
    background-color: #FA787E;
  }

  .css-form input.ng-valid.ng-touched {
    background-color: #78FA89;
  }
</style>

<script>
  angular.module('formExample', [])
  .controller('ExampleController', ['$scope', function($scope) {
   $scope.master = {};

   $scope.update = function(user) {
    $scope.master = angular.copy(user);
   };

   $scope.reset = function() {
    $scope.user = angular.copy($scope.master);
   };

   $scope.reset();
  }]);
</script>
```

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

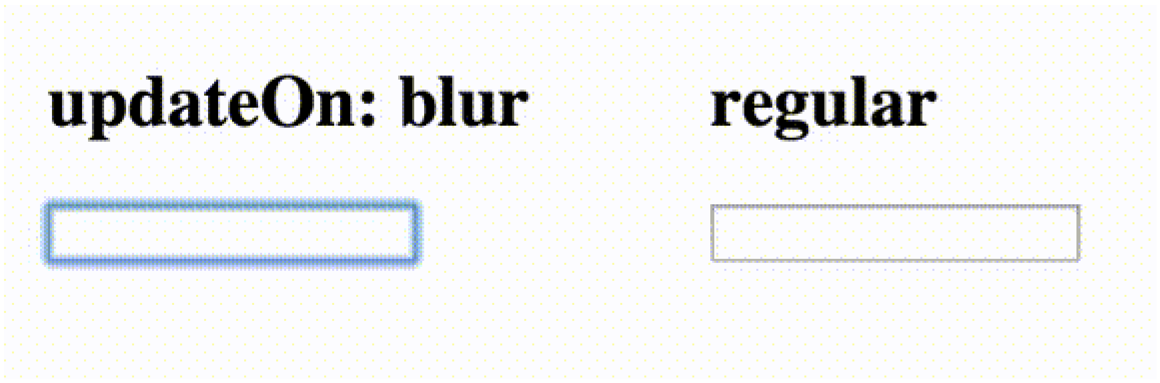
Edit in Plunker

index.htmlscript.js

```
<div ng-controller="ExampleController">
  <form name="form" class="css-form" novalidate>
    <label>Name:
      <input type="text" ng-model="user.name" name="uName" required="" />
    </label>
    <br />
    <div ng-show="form.$submitted || form.uName.$touched">
      <div ng-show="form.uName.$error.required">Tell us your name.</div>
    </div>

    <label>E-mail:
      <input type="email" ng-model="user.email" name="uEmail" required="" />
    </label>
    <br />
    <div ng-show="form.$submitted || form.uEmail.$touched">
      <span ng-show="form.uEmail.$error.required">Tell us your email.</span>
      <span ng-show="form.uEmail.$error.email">This is not a valid email.</span>
    </div>

    Gender:
    <label><input type="radio" ng-model="user.gender" value="male" />male</label>
    <label><input type="radio" ng-model="user.gender" value="female" />female</label>
    <br />
    <label>
    <input type="checkbox" ng-model="user.agree" name="userAgree" required="" />
    I agree:
    </label>
    <input ng-show="user.agree" type="text" ng-model="user.agreeSign" required="" />
    <br />
    <div ng-show="form.$submitted || form.userAgree.$touched">
      <div ng-show="!user.agree || !user.agreeSign">Please agree and sign.</div>
    </div>

    <input type="button" ng-click="reset(form)" value="Reset" />
    <input type="submit" ng-click="update(user)" value="Save" />
  </form>
  <pre>user = {{user | json}}</pre>
  <pre>master = {{master | json}}</pre>
</div>
```

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).

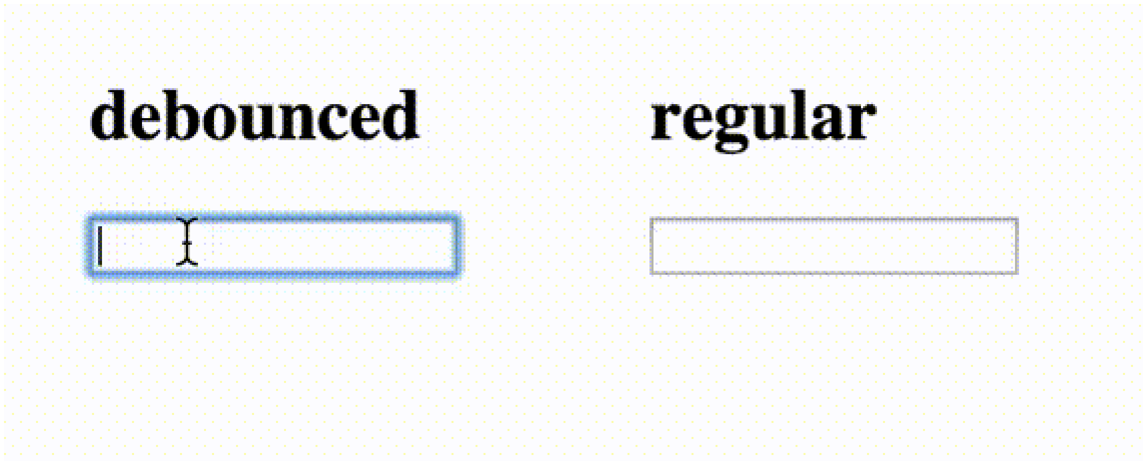
Edit in Plunker

index.htmlscript.js

```
<div ng-controller="ExampleController">
  <form>
    <label>Name:
      <input type="text" ng-model="user.name" ng-model-options="{ updateOn: 'blur' }" /></label><br />
    <label>
    Other data:
    <input type="text" ng-model="user.data" /></label><br />
  </form>
  <pre>username = "{{user.name}}"</pre>
  <pre>userdata = "{{user.data}}"</pre>
</div>
```

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.

Edit in Plunker

index.htmlscript.js

```
<div ng-controller="ExampleController">
  <form>
```

```
    <label>Name:
    <input type="text" ng-model="user.name" ng-model-options="{ debounce: 250 }" /></label><br />
  </form>
  <pre>username = "{{user.name}}"</pre>
</div>
```

Custom Validation

Angular 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 `$validators` object receives the `modelValue` and the `viewValue` as parameters. Angular will then call `$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.

Edit in Plunker

index.html**script.js**

```
<form name="form" class="css-form" novalidate>
  <div>
    <label>
    Size (integer 0 - 10):
    <input type="number" ng-model="size" name="size"
           min="0" max="10" integer />{{size}}</label><br />
    <span ng-show="form.size.$error.integer">The value is not a valid integer!</span>
    <span ng-show="form.size.$error.min || form.size.$error.max">
      The value must be in range 0 to 10!</span>
  </div>

  <div>
    <label>
    Username:
    <input type="text" ng-model="name" name="name" username />{{name}}</label><br />
    <span ng-show="form.name.$pending.username">Checking if this name is available...</span>
    <span ng-show="form.name.$error.username">This username is already taken!</span>
  </div>

</form>
```

Modifying built-in validators

Since Angular 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.

Edit in Plunker

index.htmlscript.js

```
<form name="form" class="css-form" novalidate>
  <div>
    <label>
      Overwritten Email:
      <input type="email" ng-model="myEmail" overwrite-email name="overwrittenEmail" />
    </label>
    <span ng-show="form.overwrittenEmail.$error.email">This email format is invalid!</span><br>
    Model: {{myEmail}}
    </div>
</form>
```

Implementing custom form controls (using `ngModel`)

Angular 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.

Edit in Plunker

index.htmlscript.js

```
<div contentEditable="true" ng-model="content" title="Click to edit">Some</div>
<pre>model = {{content}}</pre>


<style type="text/css">
  div[contentEditable] {
    cursor: pointer;
    background-color: #D0D0D0;
  }
</style>
```

Creating Custom Directives

Note: this guide is targeted towards developers who are already familiar with AngularJS basics. If you're just getting started, we recommend the tutorial first. If you're looking for the **directives API**, you can find it in the `$compile` API docs.

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.

Angular 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 Angular to use. When Angular 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 in compiled programming languages.

Matching Directives

Before we can write a directive, we need to know how Angular'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

Angular **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:

Edit in Plunker

index.htmlscript.jsprotractor.js

```
<div ng-controller="Controller">
 Hello <input ng-model='name'> <hr/>
 <span ng-bind="name"></span> <br/>
 <span ng:bind="name"></span> <br/>
 <span ng_bind="name"></span> <br/>
 <span data-ng-bind="name"></span> <br/>
 <span x-ng-bind="name"></span> <br/>
</div>
```

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, attributes, class names, as well as comments.

All of the Angular-provided directives match attribute name, tag name, comments, or class name. The following demonstrates the various ways a directive (`myDir` in this case) 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>
```

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 the normalized 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 Angular.

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:

Edit in Plunker

script.jsindex.html

```
angular.module('docsSimpleDirective', [])
.controller('Controller', ['$scope', function($scope) {
  $scope.customer = {
    name: 'Naomi',
    address: '1600 Amphitheatre'
  };
}])
.directive('myCustomer', function() {
  return {
    template: 'Name: {{customer.name}} Address: {{customer.address}}'
  };
});
```

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:

Edit in Plunker

script.jsindex.htmlmy-customer.html

```
angular.module('docsTemplateUrlDirective', [])
.controller('Controller', ['$scope', function($scope) {
  $scope.customer = {
    name: 'Naomi',
    address: '1600 Amphitheatre'
  };
}])
.directive('myCustomer', function() {
  return {
    templateUrl: 'my-customer.html'
  };
});
```

templateUrl can also be a function which returns the URL of an HTML template to be loaded and used for the directive. Angular 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.

Edit in Plunker

script.jsindex.htmlcustomer-name.htmlcustomer-address.html

```
angular.module('docsTemplateUrlDirective', [])
.controller('Controller', ['$scope', function($scope) {
  $scope.customer = {
    name: 'Naomi',
    address: '1600 Amphitheatre'
  };
}])
.directive('myCustomer', function() {
  return {
    templateUrl: function(elem, attr) {
      return 'customer-' + attr.type + '.html';
    }
  };
});
```

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'`:

Edit in Plunker

script.jsindex.htmlmy-customer.html

```
angular.module('docsRestrictDirective', [])
.controller('Controller', ['$scope', function($scope) {
  $scope.customer = {
    name: 'Naomi',
    address: '1600 Amphitheatre'
  };
}])
.directive('myCustomer', function() {
  return {
    restrict: 'E',
    templateUrl: 'my-customer.html'
  };
});
```

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:

Edit in Plunker

script.jsindex.htmlmy-customer.html

```
angular.module('docsScopeProblemExample', [])
.controller('NaomiController', ['$scope', function($scope) {
  $scope.customer = {
    name: 'Naomi',
    address: '1600 Amphitheatre'
  };
}])
.controller('IgorController', ['$scope', function($scope) {
  $scope.customer = {
    name: 'Igor',
    address: '123 Somewhere'
  };
}])
.directive('myCustomer', function() {
  return {
    restrict: 'E',
    templateUrl: 'my-customer.html'
  };
});
```

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:

Edit in Plunker

script.jsindex.htmlmy-customer-iso.html

```
angular.module('docsIsolateScopeDirective', [])
.controller('Controller', ['$scope', function($scope) {
  $scope.naomi = { name: 'Naomi', address: '1600 Amphitheatre' };
  $scope.igor = { name: 'Igor', address: '123 Somewhere' };
}])
.directive('myCustomer', function() {
  return {
    restrict: 'E',
    scope: {
      customerInfo: '=info'
    },
    templateUrl: 'my-customer-iso.html'
  };
});
```

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:

Edit in Plunker

script.jsindex.htmlmy-customer-plus-vojta.html

```
angular.module('docsIsolationExample', [])
.controller('Controller', ['$scope', function($scope) {
  $scope.naomi = { name: 'Naomi', address: '1600 Amphitheatre' };
  $scope.vojta = { name: 'Vojta', address: '3456 Somewhere Else' };
}])
```

```
.directive('myCustomer', function() {
  return {
    restrict: 'E',
    scope: {
      customerInfo: '=info'
    },
    templateUrl: 'my-customer-plus-vojta.html'
  };
});
```

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 Angular 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 `$timeout`s have completed before completing the test. We also want to remove the `$interval` if the directive is deleted so we don't introduce a memory leak.

Edit in Plunker

script.jsindex.html

```
angular.module('docsTimeDirective', [])
.controller('Controller', ['$scope', function($scope) {
  $scope.format = 'M/d/yy h:mm:ss a';
}])
.directive('myCurrentTime', ['$interval', 'dateFilter', function($interval, dateFilter) {

  function link(scope, element, attrs) {
    var format,
        timeoutId;

    function updateTime() {
      element.text(dateFilter(new Date(), format));
    }

    scope.$watch(attrs.myCurrentTime, function(value) {
      format = value;
      updateTime();
    });

    element.on('$destroy', function() {
      $interval.cancel(timeoutId);
    });

    // start the UI update process; save the timeoutId for canceling
    timeoutId = $interval(function() {
      updateTime(); // update DOM
    }, 1000);
  }

  return {
    link: link
  };
}]);
```

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 Angular'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.

Edit in Plunker

script.jsindex.htmlmy-dialog.html

```
angular.module('docsTransclusionDirective', [])
.controller('Controller', ['$scope', function($scope) {
  $scope.name = 'Tobias';
}])
.directive('myDialog', function() {
  return {
    restrict: 'E',
    transclude: true,
    scope: {},
    templateUrl: 'my-dialog.html'
  };
});
```

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?

Edit in Plunker

script.jsindex.htmlmy-dialog.html

```
angular.module('docsTransclusionExample', [])
.controller('Controller', ['$scope', function($scope) {
  $scope.name = 'Tobias';
}])
.directive('myDialog', function() {
  return {
    restrict: 'E',
    transclude: true,
    scope: {},
    templateUrl: 'my-dialog.html',
    link: function(scope) {
      scope.name = 'Jeff';
    }
  };
});
```

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.

Edit in Plunker

script.jsindex.htmlmy-dialog-close.html

```
angular.module('docsIsoFnBindExample', [])
.controller('Controller', ['$scope', '$timeout', function($scope, $timeout) {
  $scope.name = 'Tobias';
  $scope.message = '';
  $scope.hideDialog = function(message) {
    $scope.message = message;
    $scope.dialogIsHidden = true;
    $timeout(function() {
      $scope.message = '';
      $scope.dialogIsHidden = false;
    }, 2000);
  };
}])
.directive('myDialog', function() {
  return {
    restrict: 'E',
    transclude: true,
    scope: {
      'close': '&onClose'
    },
    templateUrl: 'my-dialog-close.html'
  };
});
```

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?

Edit in Plunker

script.jsindex.html

```
angular.module('dragModule', [])
.directive('myDraggable', ['$document', function($document) {
  return {
    link: function(scope, element, attr) {
      var startX = 0, startY = 0, x = 0, y = 0;
```

```
        element.css({
         position: 'relative',
         border: '1px solid red',
         backgroundColor: 'lightgrey',
         cursor: 'pointer'
        });

        element.on('mousedown', function(event) {
          // Prevent default dragging of selected content
          event.preventDefault();
          startX = event.pageX - x;
          startY = event.pageY - y;
          $document.on('mousemove', mousemove);
          $document.on('mouseup', mouseup);
        });

        function mousemove(event) {
          y = event.pageY - startY;
          x = event.pageX - startX;
          element.css({
            top: y + 'px',
            left:  x + 'px'
          });
        }

        function mouseup() {
          $document.off('mousemove', mousemove);
          $document.off('mouseup', mouseup);
        }
      }
    };
  }]);
```

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.

Edit in Plunker

script.jsindex.htmlmy-tabs.htmlmy-pane.html

```
angular.module('docsTabsExample', [])
.directive('myTabs', function() {
  return {
    restrict: 'E',
    transclude: true,
    scope: {},
    controller: ['$scope', function MyTabsController($scope) {
      var panes = $scope.panes = [];

      $scope.select = function(pane) {
        angular.forEach(panes, function(pane) {
          pane.selected = false;
        });
        pane.selected = true;
      };

      this.addPane = function(pane) {
        if (panes.length === 0) {
          $scope.select(pane);
        }
        panes.push(pane);
      };
    }],
    templateUrl: 'my-tabs.html'
  };
})
.directive('myPane', function() {
  return {
    require: '^^myTabs',
    restrict: 'E',
    transclude: true,
    scope: {
      title: '@'
    },
    link: function(scope, element, attrs, tabsCtrl) {
      tabsCtrl.addPane(scope);
    },
    templateUrl: 'my-pane.html'
  };
});
```

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.

Understanding Components

In Angular, 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 Angular 2'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 2

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 Angular 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.)

Index.js

```
angular.module('heroApp', []).controller('MainCtrl', function MainCtrl() {
  this.hero = {
    name: 'Spawn'
  };
});
```

heroDetail.js

```
function HeroDetailController() {


}
```

```
angular.module('heroApp').component('heroDetail', {
  templateUrl: 'heroDetail.html',
  controller: HeroDetailController,
  bindings: {
    hero: '='
  }
});
```

Index.html:

```
<!-- components match only elements -->
<div ng-controller="MainCtrl as ctrl">
  <b>Hero</b><br>
  <hero-detail hero="ctrl.hero"></hero-detail>
</div>
<span>Name: {{$ctrl.hero.name}}</span>
```

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 if 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 when it's 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, and https://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 within your application.

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

What is a Module?

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

Why?

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

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

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

The Basics

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

Edit in Plunker

index.htmlscript.jsprotractor.js

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

Script.js:

```
// declare a module
var myAppModule = angular.module('myApp', []);


// configure the module.
// in this example we will create a greeting filter
myAppModule.filter('greet', function() {
 return function(name) {
    return 'Hello, ' + name + '!';
  };
});
it('should add Hello to the name', function() {
  expect(element(by.binding("'World' | greet")).getText()).toEqual('Hello, World!');
});
```

Important things to notice:

- The Module API

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

Recommended Setup

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

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

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

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

Module Loading & Dependencies

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

1. **Configuration blocks** - get executed during the provider registrations and configuration phase. Only providers and constants can be injected into

configuration blocks. This is to prevent accidental instantiation of services before they have been fully configured.

2. **Run blocks** - get executed after the injector is created and are used to kickstart the application. Only instances and constants can be injected into run blocks. This is to prevent further system configuration during application run time.

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

Configuration Blocks

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

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

```
// is same as

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

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

Run Blocks

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

Dependencies

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

Asynchronous Loading

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

Creation versus Retrieval

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

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

// add some directives and services
myModule.service('myService', ...);
myModule.directive('myDirective', ...);

// overwrites both myService and myDirective by creating a new module
var myModule = angular.module('myModule', []);

// throws an error because myOtherModule has yet to be defined
var myModule = angular.module('myOtherModule');
```

Unit Testing

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

Each module can only be loaded once per injector. Usually an Angular app has only one injector and modules are only loaded once. Each test has its own injector and modules are loaded multiple times.

In all of these examples we are going to assume this module definition:

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

factory('alert', function($window) {
  return function(text) {
    $window.alert(text);
  }
}).

value('salutation', 'Hello').

factory('greet', function(alert, salutation) {
  return function(name) {
    alert(salutation + ' ' + name + '!');
  }
});
```

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

```
describe('myApp', function() {
  // load application module (`greetMod`) then load a special
  // test module which overrides `$window` with a mock version,
  // so that calling `window.alert()` will not block the test
  // runner with a real alert box.
  beforeEach(module('greetMod', function($provide) {
    $provide.value('$window', {
      alert: jasmine.createSpy('alert')
    });
  }));

  // inject() will create the injector and inject the `greet` and
  // `$window` into the tests.
  it('should alert on $window', inject(function(greet, $window) {
    greet('World');
    expect($window.alert).toHaveBeenCalledWith('Hello World!');
  }));

  // this is another way of overriding configuration in the
  // tests using inline `module` and `inject` methods.
  it('should alert using the alert service', function() {
    var alertSpy = jasmine.createSpy('alert');
    module(function($provide) {
      $provide.value('alert', alertSpy);
    });
    inject(function(greet) {
      greet('World');
      expect(alertSpy).toHaveBeenCalledWith('Hello World!');
    });
  });
});
```

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 Angular 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 Angular 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 Angular module. An Angular 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 Angular application starts with a given application module, Angular 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.

Decorators in AngularJS

NOTE: This guide is targeted towards developers who are already familiar with AngularJS basics. If you're just getting started, we recommend the tutorial first.

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 Angular, 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.

What does it do?

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
purpose	allow read/write access to the current browser location
API	exposes "raw" object with properties that can be directly modified
integration with angular application life-cycle	none
seamless integration with HTML5 API	no
aware of docroot/context from which the application is loaded	no - window.location.pathname returns "/docroot/actual/path"

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

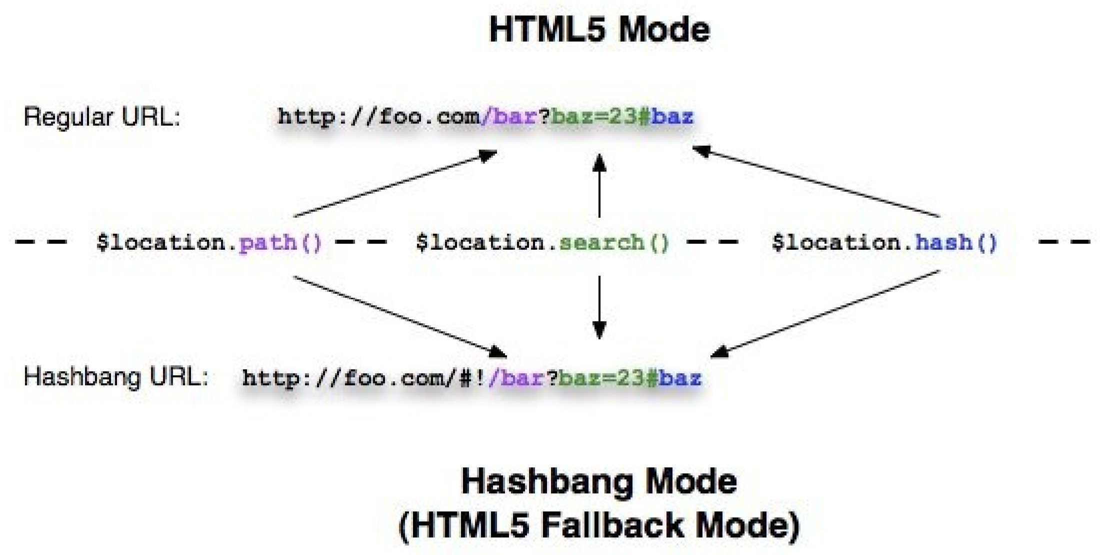
You can pass special characters to `$location` service and it will encode them according to rules specified in [RFC 3986](#). When you access the methods:

- 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 }
URL format	hashbang URLs in all browsers	regular URLs in moc

	Hashbang mode	HTML5 mode
<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. Angular 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', inject(
  function($locationProvider) {
    $locationProvider.html5Mode(false);
    $locationProvider.hashPrefix('!');
  },
  function($location) {
    // open http://example.com/base/index.html#!/a
    $location.absUrl() === 'http://example.com/base/index.html#!/a'
    $location.path() === '/a'

    $location.path('/foo')
    $location.absUrl() === 'http://example.com/base/index.html#!/foo'


    $location.search() === {}
    $location.search({a: 'b', c: true});
    $location.absUrl() === 'http://example.com/base/index.html#!/foo?a=b&c'


    $location.path('/new').search('x=y');
    $location.absUrl() === '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, Angular 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', inject(
  function($locationProvider) {
    $locationProvider.html5Mode(true);
    $locationProvider.hashPrefix('!');
  },
  function($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)
    $location.path() === '/a'

    $location.path('/foo');
    $location.absUrl() === 'http://example.com/foo'

    $location.search() === {}
    $location.search({a: 'b', c: true});
    $location.absUrl() === 'http://example.com/foo?a=b&c'

    $location.path('/new').search('x=y');
    $location.url() === 'new?x=y'
    $location.absUrl() === 'http://example.com/new?x=y'

    // 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)
    $location.path() === '/new'
    $location.search() === {x: 'y'}

    $location.path('/foo/bar');
    $location.path() === '/foo/bar'
    $location.url() === '/foo/bar?x=y'
    $location.absUrl() === '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. Angular 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 Angular 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 you 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.

i18n and l10n

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 Angular support i18n/l10n?

Angular 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 Angular components depend on locale-specific rule sets managed by the `$locale` service.

There are a few examples that showcase how to use Angular filters with various locale rule sets in the `i18n/e2e` directory of the Angular 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 Angular

Angular 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 Angular

There are two approaches to providing locale rules to Angular:

1. Pre-bundled rule sets

You can pre-bundle the desired locale file with Angular 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, Angular 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.

http://www.learn-angular.org/#!/lessons/the-essentials

http://www.w3schools.com/angular/angular_intro.asp

https://www.tutorialspoint.com/angularjs/

https://docs.angularjs.org/tutorial