What Is Angular?

AngularJS is a structural framework for dynamic web apps. It lets you use HTML as your template language and lets you extend HTML's syntax to express your application's components clearly and succinctly. Out of the box, it eliminates much of the code you currently write through data binding and dependency injection. And it all happens in JavaScript within the browser making it an ideal partner with any server technology.

Angular is what HTML would have been had it been designed for applications. HTML is a great declarative language for static documents. It does not contain much in the way of creating applications, and as a result building web applications is an exercise in *what do I have to do, so that I trick the browser in to doing what I want.*

The impedance mismatch between dynamic applications and static documents is often solved as:

- **library** - a collection of functions which are useful when writing web apps. Your code is in charge and it calls into the library when it sees fit. E.g., `jQuery`.
- **frameworks** - a particular implementation of a web application, where your code fills in the details. The framework is in charge and it calls into your code when it needs something app specific. E.g., `knockout`, `sproutcore`, etc.

Angular takes another approach. It attempts to minimize the impedance mismatch between document centric HTML and what an application needs by creating new HTML constructs. Angular teaches the browser new syntax through a construct we call directives. Examples include:

- Data binding as in `{{}}`.
- DOM control structures for repeating/hiding DOM fragments.
- Support for forms and form validation.
- Attaching code-behind to DOM elements.
- Grouping of HTML into reusable components.

End-to-end solution

Angular tries to be an end-to-end solution, when building a web application. This means it is not a single piece in an overall puzzle of building a web application, but an end-to-end solution. This makes Angular opinionated about how a CRUD application should be built. But while it is opinionated, it also tries to make sure that its opinion is just a starting point, which you can easily change. Angular comes with the following out-of-the-box:

- Everything you need to build a CRUD app in a cohesive set: data-binding, basic templating directives, form validation, routing, deep-linking, reusable components, dependency injection.
- Testability story: unit-testing, end-to-end testing, mocks, test harnesses.
- Seed application with directory layout and test scripts as a starting point.

Angular Sweet Spot

Angular simplifies application development by presenting a higher level of abstraction to the developer. Like any abstraction, it comes at a cost of flexibility. In other words not every app is a good fit for Angular. Angular was built for the CRUD application in mind. Luckily CRUD applications represent at least 90% of the web applications. But to understand what Angular is good at one also has to understand when an app is not a good fit for Angular.

Games, and GUI editors are examples of very intensive and tricky DOM manipulation. These kinds of apps are different from CRUD apps, and as a result are not a good fit for Angular. In these cases using something closer to bare metal such as `jQuery` may be a better fit.

An Introductory Angular Example

Below is a typical CRUD application which contains a form. The form values are validated, and are used to compute the total, which is formatted to a particular locale. These are some common concepts which the application developer may face:

- attaching data-model to the UI.
- writing, reading and validating user input.
- computing new values based on the model.
- formatting output in a user specific locale.

Source

index.html   script.js   End to end test   Edit

index.html :

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="InvoiceCntl">
      <b>Invoice:</b>
      <br>
      <br>
      <table>
       <tr><td>Quantity</td><td>Cost</td></tr>
       <tr>
         <td><input type="integer" min="0" ng-model="qty" required ></td>
         <td><input type="number" ng-model="cost" required ></td>
       </tr>
      </table>
      <hr>
      <b>Total:</b> {{qty * cost | currency}}
    </div>
  </body>
</html>
```

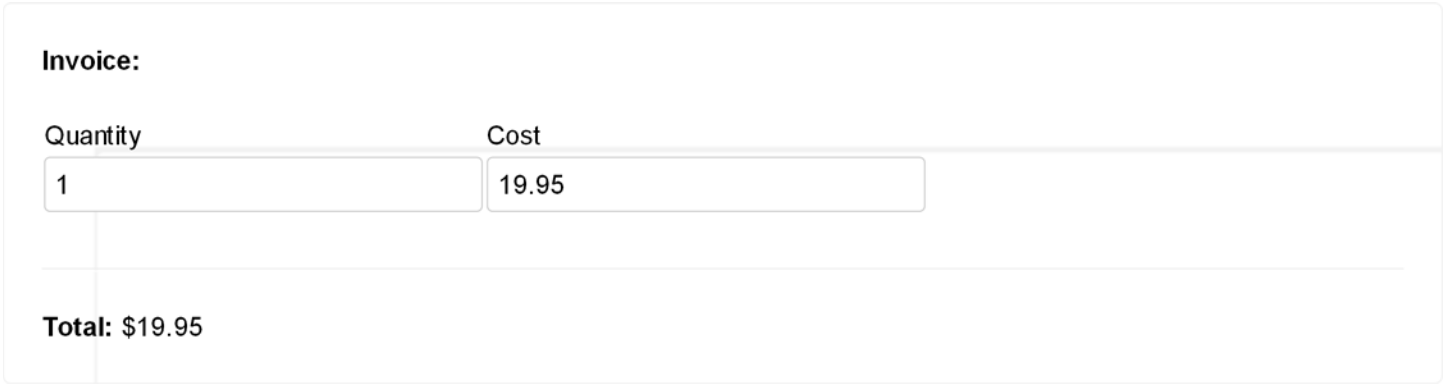
script.js :

```
function InvoiceCntl($scope) {
  $scope.qty = 1;
  $scope.cost = 19.95;
}
```

End to end test :

```
it('should show of angular binding', function() {
  expect(binding('qty * cost')).toEqual('$19.95');
  input('qty').enter('2');
  input('cost').enter('5.00');
  expect(binding('qty * cost')).toEqual('$10.00');
});
```

Demo



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

In the `<html>` tag, we specify that it is an Angular application with the `ng-app` directive. The `ng-app` will cause Angular to auto initialize your application.

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

We load Angular using the `<script>` tag:

```
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/?.?.?/angular.min.js"></script>
```

From the `ng-model` attribute of the `<input>` tags, Angular automatically sets up two-way data binding, and we also demonstrate some easy input validation:

```
Quantity: <input type="integer" min="0" ng-model="qty" required >
Cost: <input type="number" ng-model="cost" required >
```

These input widgets look normal enough, but consider these points:

- When this page loaded, Angular bound the names of the input widgets (`qty` and `cost`) to variables of the same name. Think of those variables as the "Model" component of the Model-View-Controller design pattern.
- Note that the HTML widget `input` has special powers. The input invalidates itself by turning red when you enter invalid data or leave the the input fields blank. These new widget behaviors make it easier to implement field validation common in CRUD applications.

And finally, the mysterious `{{ double curly braces }}`:

```
    Total: {{qty * cost | currency}}
```

This notation, `{{ _expression_ }}`, is Angular markup for data-binding. The expression itself can be a combination of both an expression and a filter: `{{ expression | filter }}`. Angular provides filters for formatting display data.

In the example above, the expression in double-curly braces directs Angular to "bind the data we got from the input widgets to the display, multiply them together, and format the resulting number into output that looks like money."

Notice that we achieved this application behavior not by calling Angular methods, nor by implementing application specific behavior as a framework. We achieved the behavior because the browser behaved more in line with what is needed for a dynamic web application rather then what is needed for a static document. Angular has lowered the impedance mismatch to the point where no library/framework calls are needed.

The Zen of Angular

Angular is built around the belief that declarative code is better than imperative when it comes to building UIs and wiring software components together, while imperative code is excellent for expressing business logic.

- It is a very good idea to decouple DOM manipulation from app logic. This dramatically improves the testability of the code.
- It is a really, *really* good idea to regard app testing as equal in importance to app writing. Testing difficulty is dramatically affected by the way the code is structured.
- It is an excellent idea to decouple the client side of an app from the server side. This allows development work to progress in parallel, and allows for reuse of both sides.
- It is very helpful indeed if the framework guides developers through the entire journey of building an app: from designing the UI, through writing the business logic, to testing.
- It is always good to make common tasks trivial and difficult tasks possible.

Angular frees you from the following pain:

- **Registering callbacks:** Registering callbacks clutters your code, making it hard to see the forest for the trees. Removing common boilerplate code such as callbacks is a good thing. It vastly reduces the amount of JavaScript coding *you* have to do, and it makes it easier to see what your application does.
- **Manipulating HTML DOM programmatically:** Manipulating HTML DOM is a cornerstone of AJAX applications, but it's cumbersome and error-prone. By declaratively describing how the UI should change as your application state changes, you are freed from low level DOM manipulation tasks. Most applications written with Angular never have to programmatically manipulate the DOM, although you can if you want to.
- **Marshaling data to and from the UI:** CRUD operations make up the majority of AJAX applications. The flow of marshaling data from the server to an internal object to an HTML form, allowing users to modify the form, validating the form, displaying validation errors, returning to an internal model, and then back to the server, creates a lot of boilerplate code. Angular eliminates almost all of this boilerplate, leaving code that describes the overall flow of the application rather than all of the implementation details.
- **Writing tons of initialization code just to get started:** Typically you need to write a lot of plumbing just to get a basic "Hello World" AJAX app working. With Angular you can bootstrap your app easily using services, which are auto-injected into your application in a Guice-like dependency-injection style. This allows you to get started developing features quickly. As a bonus, you get full control over the initialization process in automated tests.

Watch a Presentation About Angular

Here is a presentation on Angular from May 2012.

AngularJS MTV Meetup (May 2012)

Overview

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

- 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.
- Place `ng-app` to the root of your application, typically on the `<html>` tag if you want angular to auto-bootstrap your application.

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

- 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:`.)

  ```
  <html xmlns:ng="http://angularjs.org">
  ```

Automatic Initialization

Angular initializes automatically upon `DOMContentLoaded` event, at which point Angular looks for the `ng-app` directive which designates your application root. If the `ng-app` directive is found then Angular will:

- load the module associated with the directive.
- create the application `injector`
- compile the DOM treating the `ng-app` 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>
```

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. The example is equivalent to using the `ng-app` directive.

```
<!doctype html>
<html xmlns:ng="http://angularjs.org">
  <body>
    Hello {{'World'}}!
    <script src="http://code.angularjs.org/angular.js"></script>
    <script>
       angular.element(document).ready(function() {
         angular.bootstrap(document);
       });
    </script>
  </body>
</html>
```

This is the sequence that your code should follow:

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

Overview

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

HTML has a lot of constructs for formatting the HTML for static documents in a declarative fashion. For example if something needs to be centered, there is no need to provide instructions to the browser how the window size needs to be divided in half so that center is found, and that this center needs to be aligned with the text's center. Simply add `align="center"` attribute to any element to achieve the desired behavior. Such is the power of declarative language.

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

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

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

Compiler

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

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

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

Directive

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

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

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

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

Source

index.html script.js Edit

index.html :

```html
<!doctype html>
<html ng-app="drag">
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <span draggable>Drag ME</span>
  </body>
</html>
```

script.js :

```javascript
angular.module('drag', []).
  directive('draggable', function($document) {
    var startX=0, startY=0, x = 0, y = 0;
    return function(scope, element, attr) {
      element.css({
       position: 'relative',
       border: '1px solid red',
       backgroundColor: 'lightgrey',
       cursor: 'pointer'
      });
      element.bind('mousedown', function(event) {
        startX = event.screenX - x;
        startY = event.screenY - y;
        $document.bind('mousemove', mousemove);
        $document.bind('mouseup', mouseup);
      });

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

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

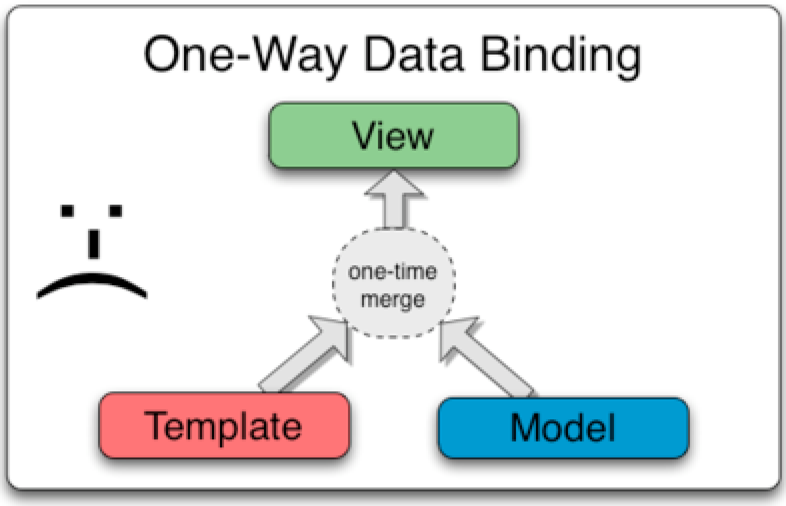
Demo

Drag ME

The presence of the `draggable` attribute on any element gives the element new behavior. The beauty of this approach is that we have taught the browser a new trick. We have extended the vocabulary of what the browser understands in a way which is natural to anyone who is familiar with HTML principles.

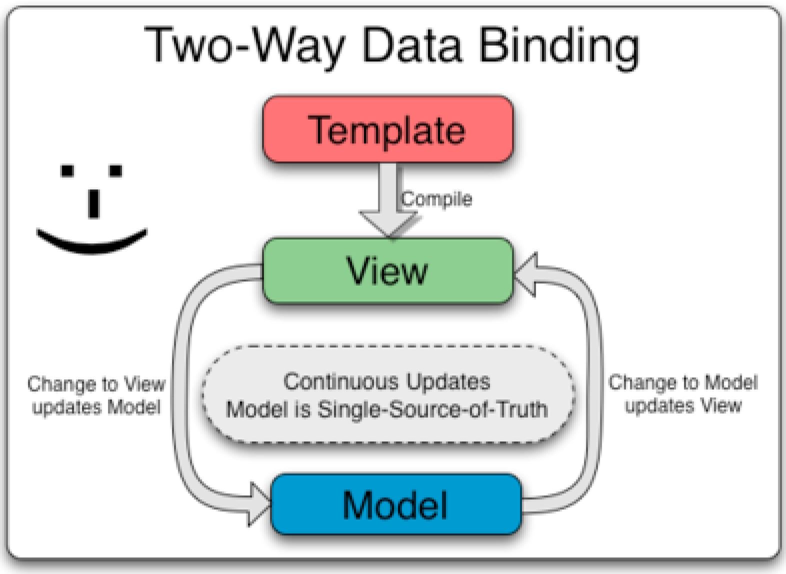
Understanding View

There are many templating systems out there. Most of them consume a static string template and combine it with data, resulting in a new string. The resulting text is then `innerHTML`ed into an element.



This means that any changes to the data need to be re-merged with the template and then `innerHTML`ed into the DOM. Some of the issues with this approach are: reading user input and merging it with data, clobbering user input by overwriting it, managing the whole update process, and lack of behavior expressiveness.

Angular is different. The Angular compiler consumes the DOM with directives, not string templates. The result is a linking function, which when combined with a scope model results in a live view. The view and scope model bindings are transparent. No action from the developer is needed to update the view. And because no `innerHTML` is used there are no issues of clobbering user input. Furthermore, Angular directives can contain not just text bindings, but behavioral constructs as well.


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

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

Overview

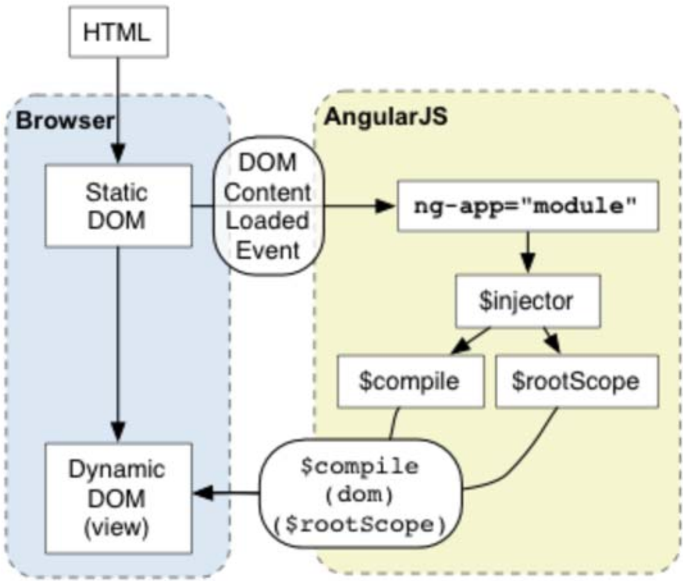
This document gives a quick overview of the main angular components and how they work together. These are:

- startup - bring up hello world
- runtime - overview of angular runtime
- scope - the glue between the view and the controller
- controller - application behavior
- model - your application data
- view - what the user sees
- directives - extend HTML vocabulary
- filters - format the data in user locale
- injector - assembles your application
- module - configures the injector
- $ - angular namespace

Startup

This is how we get the ball rolling (refer to the diagram and example below):

1. The browser loads the HTML and parses it into a DOM
2. The browser loads `angular.js` script
3. Angular waits for `DOMContentLoaded` event
4. Angular looks for `ng-app` directive, which designates the application boundary
5. The Module specified in `ng-app` (if any) is used to configure the `$injector`
6. The `$injector` is used to create the `$compile` service as well as `$rootScope`
7. The `$compile` service is used to compile the DOM and link it with `$rootScope`
8. The `ng-init` directive assigns `World` to the `name` property on the scope
9. The `{{name}}` `interpolates` the expression to `Hello World!`



Source

index.html

Edit

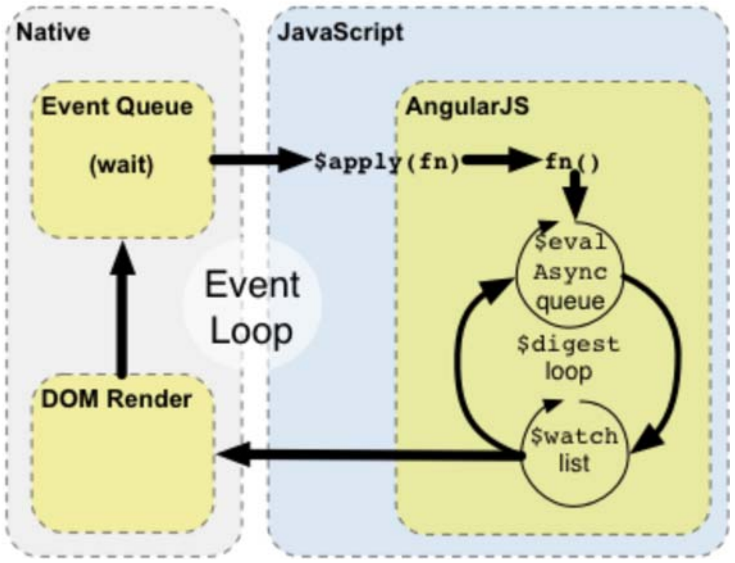
index.html :

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
  </head>
  <body>
    <p ng-init=" name='World' ">Hello {{name}}!</p>
  </body>
</html>
```

Demo

Hello World!

Runtime



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

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

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

1. Enter Angular execution context by calling `scope.$apply(stimulusFn)`. Where `stimulusFn` is the work you wish to do in Angular execution context.
2. Angular executes the `stimulusFn()`, which typically modifies application state.
3. Angular enters the `$digest` loop. The loop is made up of two smaller loops which process `$evalAsync` queue and the `$watch` list. The `$digest` loop keeps iterating until the model stabilizes, which means that the `$evalAsync` queue is empty and the `$watch` list does not detect any changes.
4. The `$evalAsync` queue is used to schedule work which needs to occur outside of current stack frame, but before the browser's view render. This is usually done with `setTimeout(0)`, but the `setTimeout(0)` approach suffers from slowness and may cause view flickering since the browser renders the view after each event.
5. The `$watch` list is a set of expressions which may have changed since last iteration. If a change is detected then the `$watch` function is called which typically updates the DOM with the new value.
6. Once the Angular `$digest` loop finishes the execution leaves the Angular and JavaScript context. This is followed by the browser re-rendering the DOM to reflect any changes.

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

1. During the compilation phase:
   1. the `ng-model` and `input` directive set up a `keydown` listener on the `<input>` control.

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

Source

index.html ✎ Edit

index.html :

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
  </head>
  <body>
    <input ng-model="name">
    <p>Hello {{name}}!</p>
  </body>
</html>
```

Demo

Hello !

Scope

The scope is responsible for detecting changes to the model section and provides the execution context for expressions. The scopes are nested in a hierarchical structure which closely follow the DOM structure. (See individual directive documentation to see which directives cause a creation of new scopes.)

The following example demonstrates how `name` expression will evaluate into different value depending on which scope it is evaluated in. The example is followed by a diagram depicting the scope boundaries.

Source

index.html style.css script.js ✎ Edit

index.html :

```html
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="GreetCtrl">
      Hello {{name}}!
    </div>
    <div ng-controller="ListCtrl">
      <ol>
        <li ng-repeat="name in names">{{name}}</li>
      </ol>
    </div>
  </body>
</html>
```

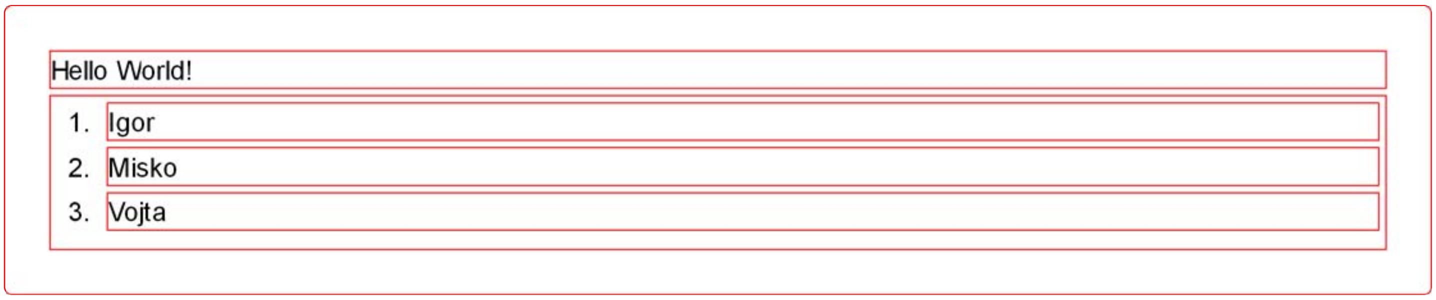
style.css :

```css
.show-scope .doc-example-live.ng-scope,
.show-scope .doc-example-live .ng-scope  {
  border: 1px solid red;
  margin: 3px;
}
```

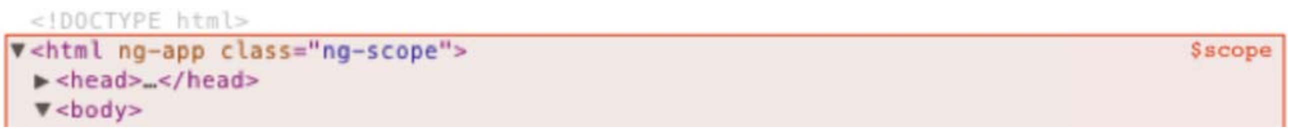
script.js :

```javascript
function GreetCtrl($scope) {
  $scope.name = 'World';
}

function ListCtrl($scope) {
  $scope.names = ['Igor', 'Misko', 'Vojta'];
}
```

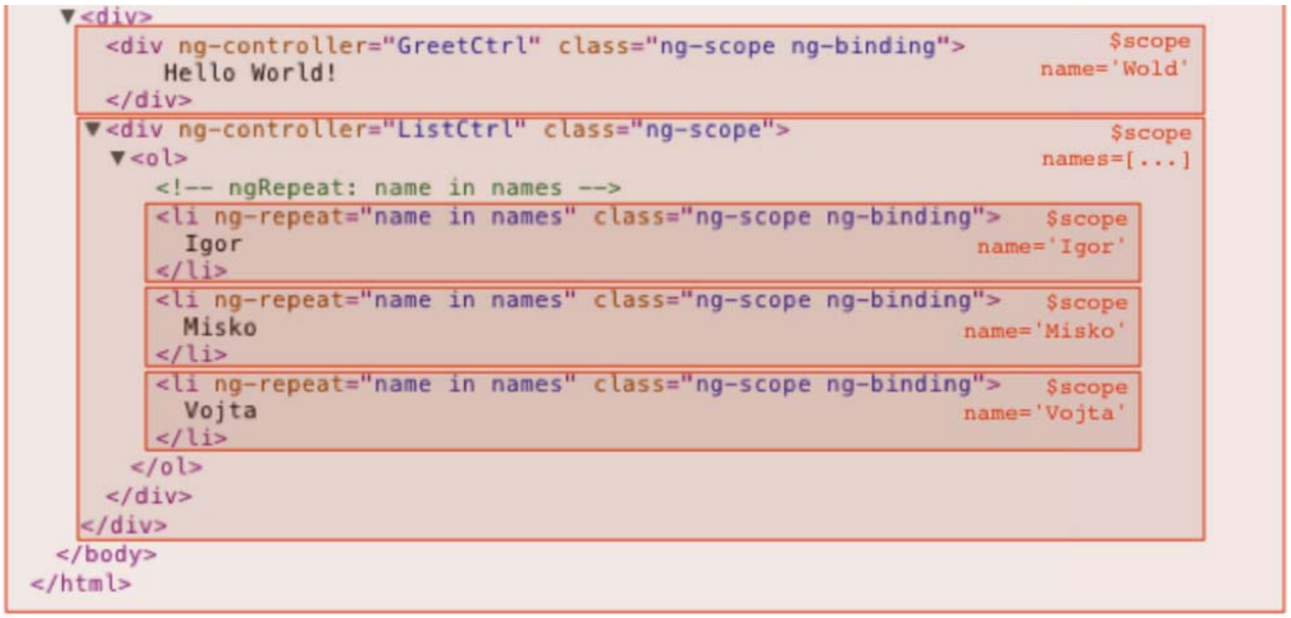
Demo



```
Hello World!
1. Igor
2. Misko
3. Vojta
```



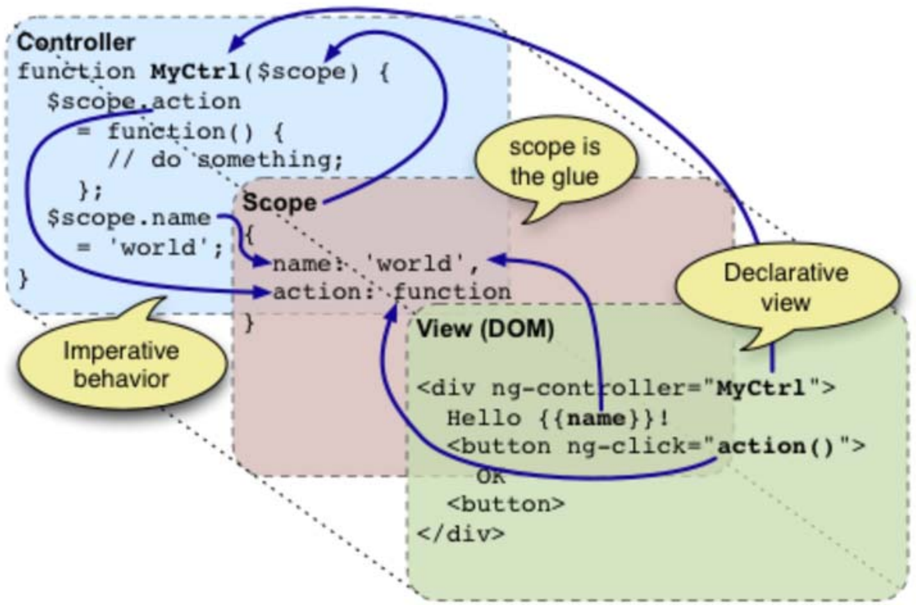
```
<!DOCTYPE html>
▼<html ng-app class="ng-scope">                                  $scope
  ▶<head>…</head>
  ▼<body>
```



```
▼<div>
   <div ng-controller="GreetCtrl" class="ng-scope ng-binding">        $scope
     Hello World!                                                  name='Wold'
   </div>
 ▼<div ng-controller="ListCtrl" class="ng-scope">                   $scope
   ▼<ol>                                                          names=[...]
      <!-- ngRepeat: name in names -->
      <li ng-repeat="name in names" class="ng-scope ng-binding">   $scope
        Igor                                                   name='Igor'
      </li>
      <li ng-repeat="name in names" class="ng-scope ng-binding">   $scope
        Misko                                                 name='Misko'
      </li>
      <li ng-repeat="name in names" class="ng-scope ng-binding">   $scope
        Vojta                                                 name='Vojta'
      </li>
    </ol>
   </div>
  </div>
 </body>
</html>
```

Controller

A controller is the code behind the view. Its job is to construct the model and publish it to the view along with callback methods. The view is a projection of the scope onto the template (the HTML). The scope is the glue which marshals the model to the view and forwards the events to the controller.



```
function MyCtrl($scope) {
  $scope.action
    = function() {
      // do something;
    };
  $scope.name
    = 'world';
}
```

```
{
  name: 'world',
  action: function
}
```

```
<div ng-controller="MyCtrl">
  Hello {{name}}!
  <button ng-click="action()">
    OK
  <button>
</div>
```

The separation of the controller and the view is important because:

- The controller is written in JavaScript. JavaScript is imperative. Imperative is a good fit for specifying application behavior. The controller should not contain any rendering information (DOM references or HTML fragments).
- The view template is written in HTML. HTML is declarative. Declarative is a good fit for specifying UI. The View should not contain any behavior.
- Since the controller is unaware of the view, there could be many views for the same controller. This is important for re-skinning, device specific views (i.e. mobile vs desktop), and testability.

Source

index.html script.js ✎ Edit

index.html :

```
<!doctype html>
<html ng-app>
```

```
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="MyCtrl">
      Hello {{name}}!
      <button ng-click="action()">
        OK
      </button>
    </div>
  </body>
</html>
```

script.js :

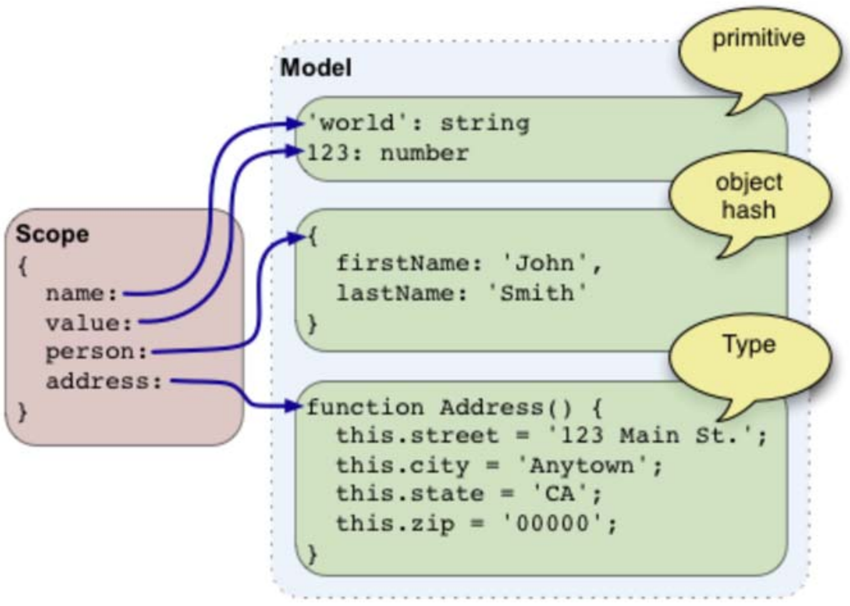
```
function MyCtrl($scope) {
  $scope.action = function() {
    $scope.name = 'OK';
  }

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

Demo

Hello World! OK

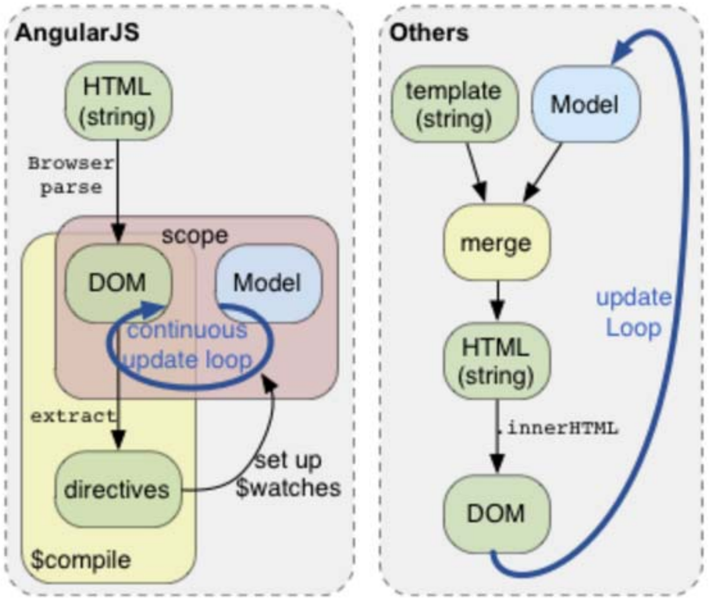
Model

The model is the data which is used merged with the template to produce the view. To be able to render the model into the view, the model has to be able to be referenced from the scope. Unlike many other frameworks Angular makes no restrictions or requirements an the model. There are no classes to inherit from or special accessor methods for accessing or changing the model. The model can be primitive, object hash, or a full object Type. In short the model is a plain JavaScript object.



View

The view is what the users sees. The view begins its life as a template, it is merged with the model and finally rendered into the browser DOM. Angular takes a very different approach to rendering the view, compared to most other templating systems.



- **Others** - Most templating systems begin as an HTML string with special templating markup. Often the template markup breaks the HTML syntax which means that the template can not be edited by an HTML editor. The template string is then parsed by the template engine, and merged with the data. The result of the merge is an HTML string. The HTML string is then written to the browser using the `.innerHTML`, which causes the browser to render the HTML. When the model changes the whole process needs to be repeated. The granularity of the template is the granularity of the DOM updates. The key here is that the templating system manipulates strings.
- **Angular** - Angular is different, since its templating system works on DOM objects not on strings. The template is still written in an HTML string, but it is HTML (not HTML with template sprinkled in.) The browser parses the HTML into the DOM, and the DOM becomes the input to the template engine known as the `compiler`. The compiler looks for directives which in turn set up `watches` on the model. The result is a continuously updating view which does not need template model re-merging. Your model becomes the single source-of-truth for your view.

Source

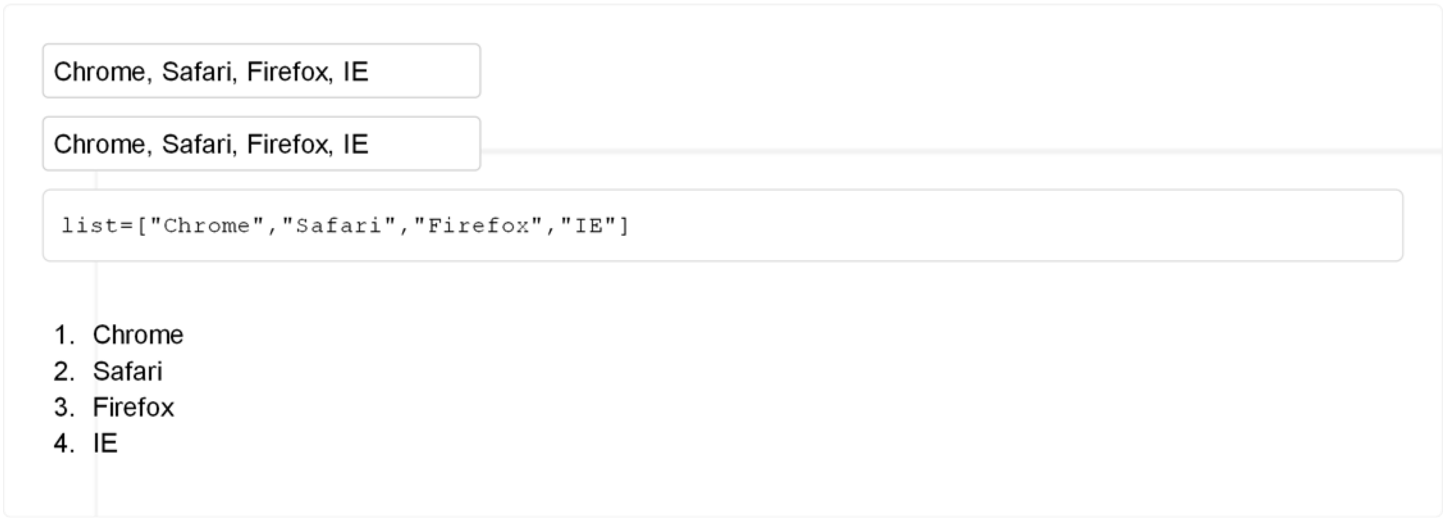
index.html

Edit

index.html :

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
  </head>
  <body>
    <div ng-init="list = ['Chrome', 'Safari', 'Firefox', 'IE'] ">
      <input ng-model="list" ng-list> <br>
      <input ng-model="list" ng-list> <br>
      <pre>list={{list}}</pre> <br>
      <ol>
        <li ng-repeat="item in list">
          {{item}}
        </li>
      </ol>
    </div>
  </body>
</html>
```

Demo



Chrome, Safari, Firefox, IE

Chrome, Safari, Firefox, IE

```
list=["Chrome","Safari","Firefox","IE"]
```

1. Chrome
2. Safari
3. Firefox
4. IE

Directives

A directive is a behavior or DOM transformation which is triggered by the presence of a custom attribute, element name, or a class name. A directive allows you to extend the HTML vocabulary in a declarative fashion. Following is an example which enables data-binding for the `contenteditable` in HTML.

Source

index.html style.css script.js Edit

index.html :

```html
<!doctype html>
<html ng-app="directive">
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div contentEditable="true" ng-model="content">Edit Me</div>
    <pre>model = {{content}}</pre>
  </body>
</html>
```

style.css :

```css
div[contentEditable] {
  cursor: pointer;
  background-color: #D0D0D0;
  margin-bottom: 1em;
  padding: 1em;
}
```

script.js :

```javascript
angular.module('directive', []).directive('contenteditable', function() {
  return {
    require: 'ngModel',
    link: function(scope, elm, attrs, ctrl) {
      // view -> model
      elm.bind('blur', function() {
        scope.$apply(function() {
          ctrl.$setViewValue(elm.html());
        });
      });

      // model -> view
      ctrl.$render = function(value) {
        elm.html(value);
      };

      // load init value from DOM
      ctrl.$setViewValue(elm.html());
    }
  };
});
```

Demo

Edit Me

```
model = Edit Me
```

Filters

`Filters` perform data transformation. Typically they are used in conjunction with the locale to format the data in locale specific output. They follow the spirit of UNIX filters and use similar syntax | (pipe).

Source

index.html

Edit

index.html :

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
  </head>
  <body>
    <div ng-init="list = ['Chrome', 'Safari', 'Firefox', 'IE'] ">
      Number formatting: {{ 1234567890 | number }} <br>
      array filtering <input ng-model="predicate">
      {{ list | filter:predicate | json }}
    </div>
  </body>
</html>
```

Demo

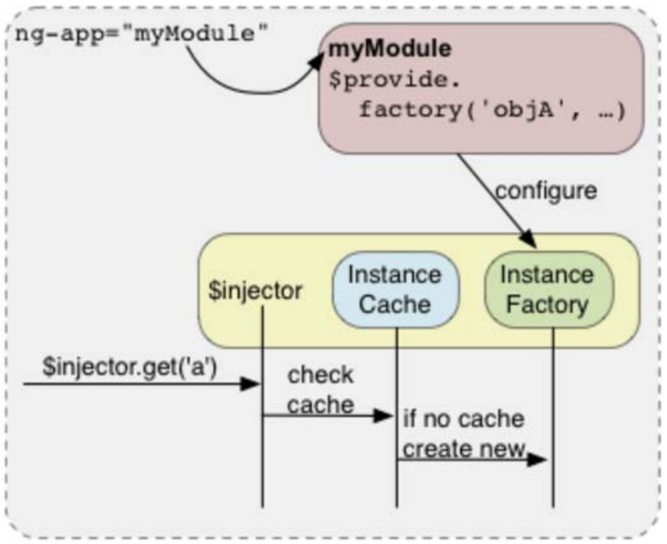
Number formatting: 1,234,567,890

array filtering ["Chrome", "Safari", "Firefox", "IE"]

Modules and the Injector

The `injector` is a service locator. There is a single `injector` per Angular `application`. The `injector` provides a way to look up an object instance by its name. The injector keeps an internal cache of all objects so that repeated calls to get the same object name result in the same instance. If the object does not exist, then the `injector` asks the instance factory to create a new instance.

A `module` is a way to configure the injector's instance factory, known as a `provider`.



```
// Create a module
var myModule = angular.module('myModule', [])

// Configure the injector
myModule.factory('serviceA', function() {
  return {
    // instead of {}, put your object creation here
  };
});

// create an injector and configure it from 'myModule'
var $injector = angular.injector(['myModule']);

// retrieve an object from the injector by name
var serviceA = $injector.get('serviceA');

// always true because of instance cache
$injector.get('serviceA') === $injector.get('serviceA');
```

But the real magic of the `injector` is that it can be used to `call` methods and `instantiate` types. This subtle feature is what allows the methods and types to ask for their dependencies instead of having to look for them.

```
// You write functions such as this one.
function doSomething(serviceA, serviceB) {
  // do something here.
}

// Angular provides the injector for your application
var $injector = ...;

///////////////////////////////////////////////
// the old-school way of getting dependencies.
var serviceA = $injector.get('serviceA');
var serviceB = $injector.get('serviceB');

// now call the function
doSomething(serviceA, serviceB);

///////////////////////////////////////////////
// the cool way of getting dependencies.
// the $injector will supply the arguments to the function automatically
$injector.invoke(doSomething); // This is how the framework calls your functions
```

Notice that the only thing you needed to write was the function, and list the dependencies in the function arguments. When angular calls the function, it will use the `call` which will automatically fill the function arguments.

Examine the `ClockCtrl` bellow, and notice how it lists the dependencies in the constructor. When the `ng-controller` instantiates the controller it automatically provides the dependencies. There is no need to create dependencies, look for dependencies, or even get a reference to the injector.

Source

index.html script.js Edit

index.html :

```
<!doctype html>
<html ng-app="timeExampleModule">
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="ClockCtrl">
      Current time is: {{ time.now }}
    </div>
  </body>
</html>
```

script.js :

```
angular.module('timeExampleModule', []).
  // Declare new object called time,
  // which will be available for injection
  factory('time', function($timeout) {
    var time = {};

    (function tick() {
      time.now = new Date().toString();
      $timeout(tick, 1000);
    })();
    return time;
  });

// Notice that you can simply ask for time
// and it will be provided. No need to look for it.
function ClockCtrl($scope, time) {
  $scope.time = time;
}
```

Demo

Current time is: Fri Jan 25 2013 09:29:59 GMT-0600 (Central Standard Time)

Angular Namespace

To prevent accidental name collision, Angular prefixes names of objects which could potentially collide with $. Please do not use the $ prefix in your code as it may accidentally collide with Angular code.

Directives are a way to teach HTML new tricks. During DOM compilation directives are matched against the HTML and executed. This allows directives to register behavior, or transform the DOM.

Angular comes with a built in set of directives which are useful for building web applications but can be extended such that HTML can be turned into a declarative domain specific language (DSL).

Invoking directives from HTML

Directives have camel cased names such as `ngBind`. The directive can be invoked by translating the camel case name into snake case with these special characters `:`, `-`, or `_`. Optionally the directive can be prefixed with `x-`, or `data-` to make it HTML validator compliant. Here is a list of some of the possible directive names: `ng:bind`, `ng-bind`, `ng_bind`, `x-ng-bind` and `data-ng-bind`.

The directives can be placed in element names, attributes, class names, as well as comments. Here are some equivalent examples of invoking `myDir`. (However, most directives are restricted to attribute only.)

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

Directives can be invoked in many different ways, but are equivalent in the end result as shown in the following example.

Source

index.html   script.js   End to end test   Edit

index.html :

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Ctrl1">
      Hello <input ng-model='name'> <hr/>
      &ltspan ng:bind="name"&gt <span ng:bind="name"></span> <br/>
      &ltspan ng_bind="name"&gt <span ng_bind="name"></span> <br/>
      &ltspan ng-bind="name"&gt <span ng-bind="name"></span> <br/>
      &ltspan data-ng-bind="name"&gt <span data-ng-bind="name"></span> <br/>
      &ltspan x-ng-bind="name"&gt <span x-ng-bind="name"></span> <br/>
    </div>
  </body>
</html>
```

script.js :

```
function Ctrl1($scope) {
  $scope.name = 'angular';
}
```

End to end test :

```
it('should show off bindings', function() {
  expect(element('div[ng-controller="Ctrl1"] span[ng-bind]').text()).toBe('angular');
});
```

Demo

Hello angular

<span ng:bind="name"> angular
<span ng_bind="name"> angular
<span ng-bind="name"> angular
<span data-ng-bind="name"> angular
<span x-ng-bind="name"> angular

String interpolation

During the compilation process the `compiler` matches text and attributes using the `$interpolate` service to see if they contain embedded expressions. These expressions are registered as `watches` and will update as part of normal `digest` cycle. An example of interpolation is shown here:

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

Compilation process, and directive matching

Compilation of HTML happens in three phases:

1. First the HTML is parsed into DOM using the standard browser API. This is important to realize because the templates must be parsable HTML. This is in contrast to most templating systems that operate on strings, rather than on DOM elements.
2. The compilation of the DOM is performed by the call to the `$compile()` method. The method traverses the DOM and matches the directives. If a match is found it is added to the list of directives associated with the given DOM element. Once all directives for a given DOM element have been identified they are sorted by priority and their `compile()` functions are executed. The directive compile function has a chance to modify the DOM structure and is responsible for producing a `link()` function explained next. The `$compile()` method returns a combined linking function, which is a collection of all of the linking functions returned from the individual directive compile functions.
3. Link the template with scope by calling the linking function returned from the previous step. This in turn will call the linking function of the individual directives allowing them to register any listeners on the elements and set up any `watches` with the `scope`. The result of this is a live binding between the scope and the DOM. A change in the scope is reflected in the DOM.

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

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

// Step 1: parse HTML into DOM element
var template = angular.element(html);

// Step 2: compile the template
var linkFn = $compile(template);

// Step 3: link the compiled template with the scope.
linkFn(scope);
```

Reasons behind the compile/link separation

At this point you may wonder why the compile process is broken down to a compile and link phase. To understand this, let's look at a real world example with a repeater:

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

The short answer is that compile and link separation is needed any time a change in model causes a change in DOM structure such as in repeaters.

When the above example is compiled, the compiler visits every node and looks for directives. The `{{user}}` is an example of an `interpolation` directive. `ngRepeat` is another directive. But `ngRepeat` has a dilemma. It needs to be able to quickly stamp out new `li`s for every `action` in `user.actions`. This means that it needs to save a clean copy of the `li` element for cloning purposes and as new `action`s are inserted, the template `li` element needs to be cloned and inserted into `ul`. But cloning the `li` element is not enough. It also needs to compile the `li` so that its directives such as `{{action.descriptions}}` evaluate against the right `scope`. A naive method would be to simply insert a copy of the `li` element and then compile it. But compiling on every `li` element clone would be slow, since the compilation requires that we traverse the DOM tree and look for directives and execute them. If we put the compilation inside a repeater which needs to unroll 100 items we would quickly run into performance problems.

The solution is to break the compilation process into two phases; the compile phase where all of the directives are identified and sorted by priority, and a linking phase where any work which links a specific instance of the `scope` and the specific instance of an `li` is performed.

`ngRepeat` works by preventing the compilation process from descending into the `li` element. Instead the `ngRepeat` directive compiles `li` separately. The result of the `li` element compilation is a linking function which contains all of the directives contained in the `li` element, ready to be attached to a specific clone of the `li` element. At runtime the `ngRepeat` watches the expression and as items are added to the array it clones the `li` element, creates a new `scope` for the cloned `li` element and calls the link function on the cloned `li`.

Summary:

- *compile function* - The compile function is relatively rare in directives, since most directives are concerned with working with a specific DOM element instance rather than transforming the template DOM element. Any operation which can be shared among the instance of directives should be moved to the compile function for performance reasons.
- *link function* - It is rare for the directive not to have a link function. A link function allows the directive to register

listeners to the specific cloned DOM element instance as well as to copy content into the DOM from the scope.

Writing directives (short version)

In this example we will build a directive that displays the current time.

Source

index.html script.js Edit

index.html :

```html
<!doctype html>
<html ng-app="time">
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Ctrl2">
      Date format: <input ng-model="format"> <hr/>
      Current time is: <span my-current-time="format"></span>
    </div>
  </body>
</html>
```

script.js :

```javascript
function Ctrl2($scope) {
  $scope.format = 'M/d/yy h:mm:ss a';
}

angular.module('time', [])
  // Register the 'myCurrentTime' directive factory method.
  // We inject $timeout and dateFilter service since the factory method is DI.
  .directive('myCurrentTime', function($timeout, dateFilter) {
    // return the directive link function. (compile function not needed)
    return function(scope, element, attrs) {
      var format,  // date format
          timeoutId; // timeoutId, so that we can cancel the time updates

      // used to update the UI
      function updateTime() {
        element.text(dateFilter(new Date(), format));
      }

      // watch the expression, and update the UI on change.
      scope.$watch(attrs.myCurrentTime, function(value) {
        format = value;
        updateTime();
      });

      // schedule update in one second
      function updateLater() {
        // save the timeoutId for canceling
        timeoutId = $timeout(function() {
          updateTime(); // update DOM
          updateLater(); // schedule another update
        }, 1000);
      }

      // listen on DOM destroy (removal) event, and cancel the next UI update
      // to prevent updating time ofter the DOM element was removed.
      element.bind('$destroy', function() {
        $timeout.cancel(timeoutId);
      });

      updateLater(); // kick off the UI update process.
    }
  });
```

Demo

Date format: M/d/yy h:mm:ss a

Current time is: 1/25/13 9:30:19 AM

Writing directives (long version)

An example skeleton of the directive is shown here, for the complete list see below.

```
var myModule = angular.module(...);

myModule.directive('directiveName', function factory(injectables) {
  var directiveDefinitionObject = {
    priority: 0,
    template: '<div></div>',
    templateUrl: 'directive.html',
    replace: false,
    transclude: false,
    restrict: 'A',
    scope: false,
    compile: function compile(tElement, tAttrs, transclude) {
      return {
        pre: function preLink(scope, iElement, iAttrs, controller) { ... },
        post: function postLink(scope, iElement, iAttrs, controller) { ... }
      }
    },
    link: function postLink(scope, iElement, iAttrs) { ... }
  };
  return directiveDefinitionObject;
});
```

In most cases you will not need such fine control and so the above can be simplified. All of the different parts of this skeleton are explained in following sections. In this section we are interested only in some of this skeleton.

The first step in simplyfing the code is to rely on the default values. Therefore the above can be simplified as:

```
var myModule = angular.module(...);

myModule.directive('directiveName', function factory(injectables) {
  var directiveDefinitionObject = {
    compile: function compile(tElement, tAttrs) {
      return function postLink(scope, iElement, iAttrs) { ... }
    }
  };
  return directiveDefinitionObject;
});
```

Most directives concern themselves only with instances, not with template transformations, allowing further simplification:

```
var myModule = angular.module(...);

myModule.directive('directiveName', function factory(injectables) {
  return function postLink(scope, iElement, iAttrs) { ... }
});
```

Factory method

The factory method is responsible for creating the directive. It is invoked only once, when the `compiler` matches the directive for the first time. You can perform any initialization work here. The method is invoked using the `$injector.invoke` which makes it injectable following all of the rules of injection annotation.

Directive Definition Object

The directive definition object provides instructions to the `compiler`. The attributes are:

- `name` - Name of the current scope. Optional and defaults to the name at registration.
- `priority` - When there are multiple directives defined on a single DOM element, sometimes it is necessary to specify the order in which the directives are applied. The `priority` is used to sort the directives before their `compile` functions get called. Higher `priority` goes first. The order of directives within the same priority is undefined.
- `terminal` - If set to true then the current `priority` will be the last set of directives which will execute (any directives at the current priority will still execute as the order of execution on same `priority` is undefined).
- `scope` - If set to:
  - `true` - then a new scope will be created for this directive. If multiple directives on the same element request a new scope, only one new scope is created. The new scope rule does not apply for the root of the template since the root of the template always gets a new scope.
  - `{}` (object hash) - then a new 'isolate' scope is created. The 'isolate' scope differs from normal scope in that it does not prototypically inherit from the parent scope. This is useful when creating reusable components, which should not accidentally read or modify data in the parent scope.

The 'isolate' scope takes an object hash which defines a set of local scope properties derived from the parent scope. These local properties are useful for aliasing values for templates. Locals definition is a hash of local scope property to its source:

  - `@` or `@attr` - bind a local scope property to the value of DOM attribute. The result is always a string since DOM attributes are strings. If no `attr` name is specified then the attribute name is assumed to be the same as the local name. Given `<widget my-attr="hello {{name}}">` and widget definition of `scope: { localName:'@myAttr' }`, then widget scope property `localName` will reflect the interpolated value of `hello {{name}}`. As the `name` attribute changes so will the `localName` property on the widget scope. The `name` is read from the parent scope (not component scope).

  - `=` or `=attr` - set up bi-directional binding between a local scope property and the parent scope property of name defined via the value of the `attr` attribute. If no `attr` name is specified then the attribute name is assumed to be the same as the local name. Given `<widget my-attr="parentModel">` and widget definition of `scope: { localModel:'=myAttr' }`, then widget scope property `localModel` will reflect the value of `parentModel` on the parent scope. Any changes to `parentModel` will be reflected in `localModel` and any changes in `localModel` will reflect in `parentModel`.

  - `&` or `&attr` - provides a way to execute an expression in the context of the parent scope. If no `attr` name is specified then the attribute name is assumed to be the same as the local name. Given `<widget my-attr="count = count + value">` and widget definition of `scope: { localFn:'&myAttr' }`, then isolate scope property `localFn` will point to a function wrapper for the `count = count + value` expression. Often it's desirable to pass data from the isolated scope via an expression and to the parent scope, this can be done by passing a map of local variable names and values into the expression wrapper fn. For example, if the expression is `increment(amount)` then we can specify the amount value by calling the `localFn` as `localFn({amount: 22})`.

- `controller` - Controller constructor function. The controller is instantiated before the pre-linking phase and it is shared with other directives if they request it by name (see `require` attribute). This allows the directives to communicate with each other and augment each other's behavior. The controller is injectable with the following locals:
  - `$scope` - Current scope associated with the element
  - `$element` - Current element
  - `$attrs` - Current attributes obeject for the element
  - `$transclude` - A transclude linking function pre-bound to the correct transclusion scope: `function(cloneLinkingFn)`.
- `require` - Require another controller be passed into current directive linking function. The `require` takes a name of the directive controller to pass in. If no such controller can be found an error is raised. The name can be prefixed with:
  - `?` - Don't raise an error. This makes the require dependency optional.
  - `^` - Look for the controller on parent elements as well.
- `restrict` - String of subset of `EACM` which restricts the directive to a specific directive declaration style. If omitted directives are allowed on attributes only.
  - `E` - Element name: `<my-directive></my-directive>`
  - `A` - Attribute: `<div my-directive="exp"> </div>`
  - `C` - Class: `<div class="my-directive: exp;"></div>`
  - `M` - Comment: `<!-- directive: my-directive exp -->`
- `template` - replace the current element with the contents of the HTML. The replacement process migrates all of the attributes / classes from the old element to the new one. See Creating Widgets section below for more information.

- `templateUrl` - Same as `template` but the template is loaded from the specified URL. Because the template loading is asynchronous the compilation/linking is suspended until the template is loaded.

- `replace` - if set to `true` then the template will replace the current element, rather than append the template to the element.

- `transclude` - compile the content of the element and make it available to the directive. Typically used with `ngTransclude`. The advantage of transclusion is that the linking function receives a transclusion function which is pre-bound to the correct scope. In a typical setup the widget creates an `isolate` scope, but the transclusion is not a child, but a sibling of the `isolate` scope. This makes it possible for the widget to have private state, and the transclusion to be bound to the parent (pre-`isolate`) scope.
  - `true` - transclude the content of the directive.
  - `'element'` - transclude the whole element including any directives defined at lower priority.
- `compile`: This is the compile function described in the section below.
- `link`: This is the link function described in the section below. This property is used only if the `compile` property is not defined.

Compile function

```
function compile(tElement, tAttrs, transclude) { ... }
```

The compile function deals with transforming the template DOM. Since most directives do not do template transformation, it is not used often. Examples that require compile functions are directives that transform template DOM, such as `ngRepeat`, or load the contents asynchronously, such as `ngView`. The compile function takes the following arguments.

- `tElement` - template element - The element where the directive has been declared. It is safe to do template transformation on the element and child elements only.
- `tAttrs` - template attributes - Normalized list of attributes declared on this element shared between all directive compile functions. See Attributes.
- `transclude` - A transclude linking function: `function(scope, cloneLinkingFn)`.

NOTE: The template instance and the link instance may not be the same objects if the template has been cloned. For this reason it is not safe in the compile function to do anything other than DOM transformation that applies to all DOM clones. Specifically, DOM listener registration should be done in a linking function rather than in a compile function.

A compile function can have a return value which can be either a function or an object.

- returning a function - is equivalent to registering the linking function via the `link` property of the config object when the compile function is empty.
- returning an object with function(s) registered via `pre` and `post` properties - allows you to control when a linking function should be called during the linking phase. See info about pre-linking and post-linking functions below.

Linking function

```
function link(scope, iElement, iAttrs, controller) { ... }
```

The link function is responsible for registering DOM listeners as well as updating the DOM. It is executed after the template has been cloned. This is where most of the directive logic will be put.

- `scope` - `Scope` - The scope to be used by the directive for registering `watches`.
- `iElement` - instance element - The element where the directive is to be used. It is safe to manipulate the children of the element only in `postLink` function since the children have already been linked.
- `iAttrs` - instance attributes - Normalized list of attributes declared on this element shared between all directive linking functions. See Attributes.

- `controller` - a controller instance - A controller instance if at least one directive on the element defines a controller. The controller is shared among all the directives, which allows the directives to use the controllers as a communication channel.

Pre-linking function
Executed before the child elements are linked. Not safe to do DOM transformation since the compiler linking function will fail to locate the correct elements for linking.

Post-linking function
Executed after the child elements are linked. It is safe to do DOM transformation in the post-linking function.

Attributes

The `Attributes` object - passed as a parameter in the link() or compile() functions - is a way of accessing:

- *normalized attribute names:* Since a directive such as 'ngBind' can be expressed in many ways such as 'ng:bind', or 'x-ng-bind', the attributes object allows for normalized accessed to the attributes.
- *directive inter-communication:* All directives share the same instance of the attributes object which allows the directives to use the attributes object as inter directive communication.
- *supports interpolation:* Interpolation attributes are assigned to the attribute object allowing other directives to read the interpolated value.
- *observing interpolated attributes:* Use `$observe` to observe the value changes of attributes that contain interpolation (e.g. `src="{{bar}}"`). Not only is this very efficient but it's also the only way to easily get the actual value because during the linking phase the interpolation hasn't been evaluated yet and so the value is at this time set to `undefined`.

```
function linkingFn(scope, elm, attrs, ctrl) {
  // get the attribute value
  console.log(attrs.ngModel);

  // change the attribute
  attrs.$set('ngModel', 'new value');

  // observe changes to interpolated attribute
  attrs.$observe('ngModel', function(value) {
    console.log('ngModel has changed value to ' + value);
  });
}
```

Understanding Transclusion and Scopes

It is often desirable to have reusable components. Below is a pseudo code showing how a simplified dialog component may work.

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

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

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

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

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

The first issue we have to solve is that the dialog box template expects `title` to be defined, but the place of instantiation would like to bind to `username`. Furthermore the buttons expect the `onOk` and `onCancel` functions to be present in the scope. This limits the usefulness of the widget. To solve the mapping issue we use the `locals` to create local variables which the template expects as follows:

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

Creating local properties on widget scope creates two problems:

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

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

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

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

Therefore the final directive definition looks something like this:

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

Creating Components

It is often desirable to replace a single directive with a more complex DOM structure. This allows the directives to become a short hand for reusable components from which applications can be built.

Following is an example of building a reusable widget.

Source

index.html   style.css   script.js   End to end test   Edit

index.html :

```
<!doctype html>
<html ng-app="zippyModule">
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Ctrl3">
      Title: <input ng-model="title"> <br>
      Text: <textarea ng-model="text"></textarea>
      <hr>
      <div class="zippy" zippy-title="Details: {{title}}...">{{text}}</div>
    </div>
  </body>
</html>
```

style.css :

```
.zippy {
  border: 1px solid black;
  display: inline-block;
  width: 250px;
}
.zippy.opened > .title:before { content: '▼ '; }
.zippy.opened > .body { display: block; }
```

```css
.zippy.closed > .title:before { content: '► '; }
.zippy.closed > .body { display: none; }
.zippy > .title {
  background-color: black;
  color: white;
  padding: .1em .3em;
  cursor: pointer;
}
.zippy > .body {
  padding: .1em .3em;
}
```

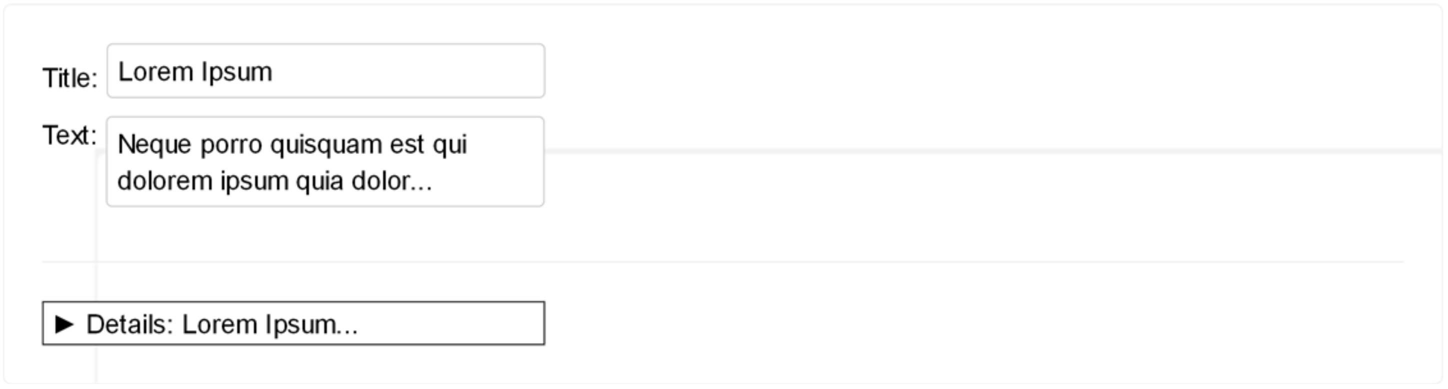
script.js :

```
function Ctrl3($scope) {
  $scope.title = 'Lorem Ipsum';
  $scope.text = 'Neque porro quisquam est qui dolorem ipsum quia dolor...';
}

angular.module('zippyModule', [])
  .directive('zippy', function(){
    return {
      restrict: 'C',
      // This HTML will replace the zippy directive.
      replace: true,
      transclude: true,
      scope: { title:'@zippyTitle' },
      template: '<div>' +
                  '<div class="title">{{title}}</div>' +
                  '<div class="body" ng-transclude></div>' +
                '</div>',
      // The linking function will add behavior to the template
      link: function(scope, element, attrs) {
            // Title element
        var title = angular.element(element.children()[0]),
            // Opened / closed state
            opened = true;

        // Clicking on title should open/close the zippy
        title.bind('click', toggle);

        // Toggle the closed/opened state
        function toggle() {
          opened = !opened;
          element.removeClass(opened ? 'closed' : 'opened');
          element.addClass(opened ? 'opened' : 'closed');
        }

        // initialize the zippy
        toggle();
      }
    }
  });
```

End to end test :

```
it('should bind and open / close', function() {
  input('title').enter('TITLE');
  input('text').enter('TEXT');
  expect(element('.title').text()).toEqual('Details: TITLE...');
  expect(binding('text')).toEqual('TEXT');

  expect(element('.zippy').prop('className')).toMatch(/closed/);
  element('.zippy > .title').click();
  expect(element('.zippy').prop('className')).toMatch(/opened/);
});
```

Demo



Title: Lorem Ipsum

Text: Neque porro quisquam est qui dolorem ipsum quia dolor...

► Details: Lorem Ipsum...

Expressions are JavaScript-like code snippets that are usually placed in bindings such as `{{ expression }}`. Expressions are processed by `$parse` service.

For example, these are all valid expressions in angular:

- `1+2`
- `3*10 | currency`
- `user.name`

Angular Expressions vs. JS Expressions

It might be tempting to think of Angular view expressions as JavaScript expressions, but that is not entirely correct, since Angular does not use a JavaScript `eval()` to evaluate expressions. You can think of Angular expressions as JavaScript expressions with following differences:

- **Attribute Evaluation:** evaluation of all properties are against the scope, doing the evaluation, unlike in JavaScript where the expressions are evaluated against the global `window`.
- **Forgiving:** expression evaluation is forgiving to undefined and null, unlike in JavaScript, where such evaluations generate `NullPointerExceptions`.
- **No Control Flow Statements:** you cannot do any of the following in angular expression: conditionals, loops, or throw.
- **Filters:** you can pass result of expression evaluations through filter chains. For example to convert date object into a local specific human-readable format.

If, on the other hand, you do want to run arbitrary JavaScript code, you should make it a controller method and call the method. If you want to `eval()` an angular expression from JavaScript, use the `$eval()` method.

Example

Source

index.html End to end test Edit

index.html :

```html
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
  </head>
  <body>
    1+2={{1+2}}
  </body>
</html>
```

End to end test :

```javascript
it('should calculate expression in binding', function() {
  expect(binding('1+2')).toEqual('3');
});
```

Demo

1+2=3

You can try evaluating different expressions here:

Source

index.html script.js End to end test Edit

index.html :

```html
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Cntl2" class="expressions">
      Expression:
      <input type='text' ng-model="expr" size="80"/>
      <button ng-click="addExp(expr)">Evaluate</button>
      <ul>
       <li ng-repeat="expr in exprs">
         [ <a href="" ng-click="removeExp($index)">X</a> ]
         <tt>{{expr}}</tt> => <span ng-bind="$parent.$eval(expr)"></span>
        </li>
      </ul>
    </div>
  </body>
</html>
```

script.js :

```javascript
function Cntl2($scope) {
  var exprs = $scope.exprs = [];
  $scope.expr = '3*10|currency';
  $scope.addExp = function(expr) {
     exprs.push(expr);
  };

  $scope.removeExp = function(index) {
    exprs.splice(index, 1);
  };
}
```

End to end test :

```javascript
it('should allow user expression testing', function() {
   element('.expressions :button').click();
```

```
    var li = using('.expressions ul').repeater('li');
    expect(li.count()).toBe(1);
    expect(li.row(0)).toEqual(["3*10|currency", "$30.00"]);
});
```

Demo

Expression: 3*10|currency Evaluate

Property Evaluation

Evaluation of all properties takes place against a scope. Unlike JavaScript, where names default to global window properties, Angular expressions have to use `$window` to refer to the global `window` object. For example, if you want to call `alert()`, which is defined on `window`, in an expression you must use `$window.alert()`. This is done intentionally to prevent accidental access to the global state (a common source of subtle bugs).

Source

index.html script.js End to end test Edit

index.html :

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div class="example2" ng-controller="Cntl1">
      Name: <input ng-model="name" type="text"/>
      <button ng-click="greet()">Greet</button>
    </div>
  </body>
</html>
```

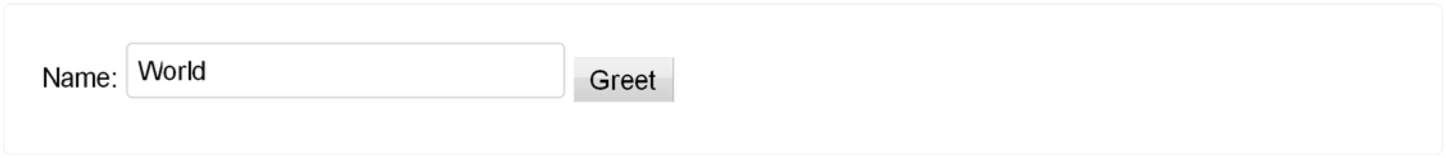
script.js :

```
function Cntl1($window, $scope){
  $scope.name = 'World';

  $scope.greet = function() {
    ($window.mockWindow || $window).alert('Hello ' + $scope.name);
  }
}
```

End to end test :

```
it('should calculate expression in binding', function() {
  var alertText;
  this.addFutureAction('set mock', function($window, $document, done) {
    $window.mockWindow = {
      alert: function(text){ alertText = text; }
    };
    done();
  });
  element(':button:contains(Greet)').click();
  expect(this.addFuture('alert text', function(done) {
    done(null, alertText);
  })).toBe('Hello World');
});
```

Demo


Name: World

Forgiving

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

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

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

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

No Control Flow Statements

You cannot write a control flow statement in an expression. The reason behind this is core to the Angular philosophy that application logic should be in controllers, not in the view. If you need a conditional, loop, or to throw from a view expression, delegate to a JavaScript method instead.

Filters

When presenting data to the user, you might need to convert the data from its raw format to a user-friendly format. For example, you might have a data object that needs to be formatted according to the locale before displaying it to the user. You can pass expressions through a chain of filters like this:

```
name | uppercase
```

The expression evaluator simply passes the value of name to `uppercase` filter.

Chain filters using this syntax:

```
value | filter1 | filter2
```

You can also pass colon-delimited arguments to filters, for example, to display the number 123 with 2 decimal points:

```
123 | number:2
```

The $

You might be wondering, what is the significance of the `$` prefix? It is simply a prefix that angular uses, to differentiate its API names from others. If angular didn't use `$`, then evaluating `a.length()` would return undefined because neither a nor angular define such a property.

Consider that in a future version of Angular we might choose to add a length method, in which case the behavior of the expression would change. Worse yet, you the developer could create a length property and then we would have a collision. This problem exists because Angular augments existing objects with additional behavior. By prefixing its additions with `$` we are reserving our namespace so that angular developers and developers who use Angular can develop in harmony without collisions.

Controls (`input`, `select`, `textarea`) are a way for user to enter data. 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. This provides a better user experience, 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.

Source

index.html script.js Edit

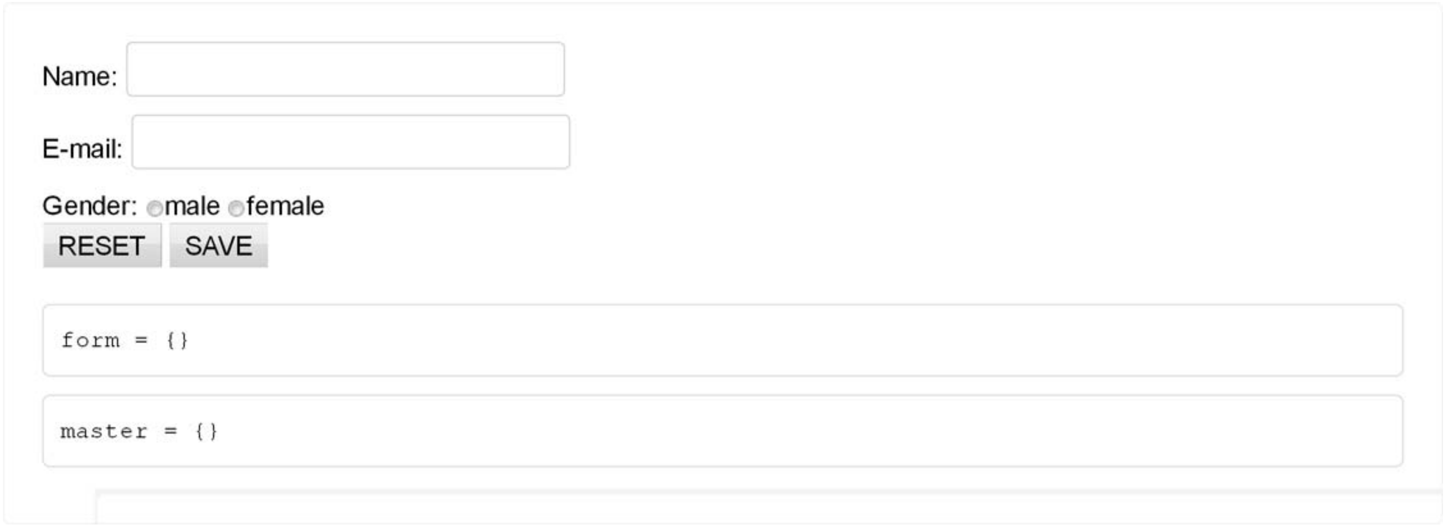
index.html :

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Controller">
      <form novalidate class="simple-form">
        Name: <input type="text" ng-model="user.name" /><br />
        E-mail: <input type="email" ng-model="user.email" /><br />
        Gender: <input type="radio" ng-model="user.gender" value="male" />male
        <input type="radio" ng-model="user.gender" value="female" />female<br />
        <button ng-click="reset()">RESET</button>
        <button ng-click="update(user)">SAVE</button>
      </form>
      <pre>form = {{user | json}}</pre>
      <pre>master = {{master | json}}</pre>
    </div>
  </body>
</html>
```

script.js :

```
function Controller($scope) {
  $scope.master= {};

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

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

  $scope.reset();
}
```

Demo



```
form = {}
```

```
master = {}
```

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

Using CSS classes

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

- `ng-valid`
- `ng-invalid`
- `ng-pristine`
- `ng-dirty`

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 when they are dirty. This ensures that the user is not distracted with an error until after interacting with the control, and failing to satisfy its validity.

Source

index.html script.js Edit

index.html :

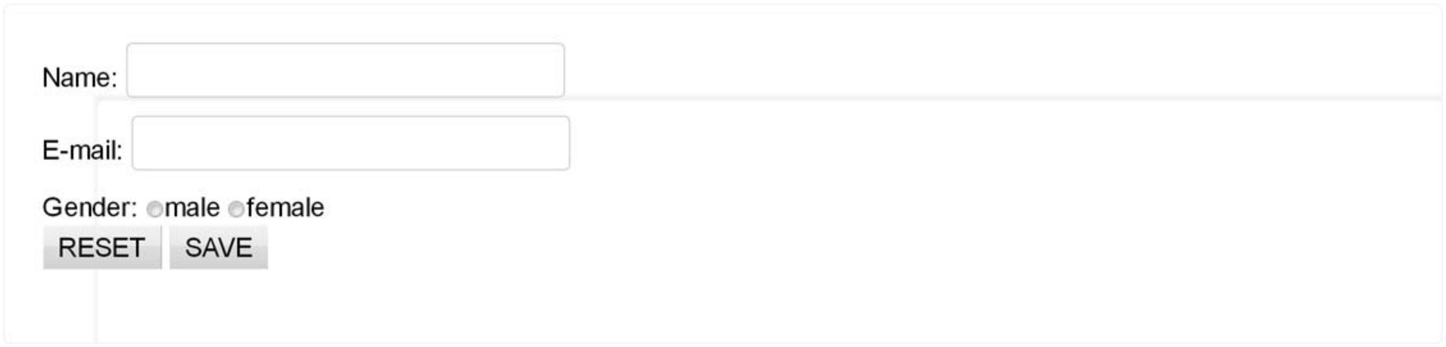
```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
    <script src="script.js"></script>
  </head>
```

```html
  <body>
    <div ng-controller="Controller">
      <form novalidate class="css-form">
        Name:
          <input type="text" ng-model="user.name" required /><br />
        E-mail: <input type="email" ng-model="user.email" required /><br />
        Gender: <input type="radio" ng-model="user.gender" value="male" />male
        <input type="radio" ng-model="user.gender" value="female" />female<br />
        <button ng-click="reset()">RESET</button>
        <button ng-click="update(user)">SAVE</button>
      </form>
    </div>

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

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

script.js :

```javascript
function Controller($scope) {
  $scope.master= {};

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

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

  $scope.reset();
}
```

Demo



Binding to form and control state

A form is in instance of `FormController`. The form instance can optionally be published into the scope using the `name` attribute. Similarly control is an instance of `NgModelController`. The control instance can similarly be published into the form instance using the `name` attribute. 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:

- RESET button is enabled only if form has some changes
- SAVE button is enabled only if form has some changes and is valid
- custom error messages for `user.email` and `user.agree`

Source

index.html script.js Edit

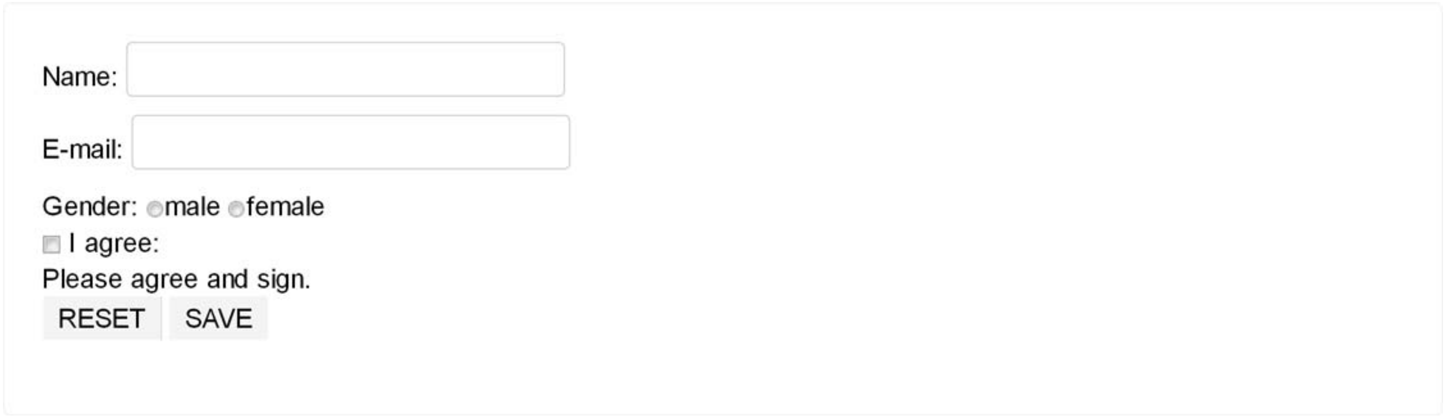
index.html :

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Controller">
      <form name="form" class="css-form" novalidate>
        Name:
          <input type="text" ng-model="user.name" name="uName" required /><br />
        E-mail:
          <input type="email" ng-model="user.email" name="uEmail" required/><br />
        <div ng-show="form.uEmail.$dirty && form.uEmail.$invalid">Invalid:
          <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: <input type="radio" ng-model="user.gender" value="male" />male
        <input type="radio" ng-model="user.gender" value="female" />female<br />

        <input type="checkbox" ng-model="user.agree" name="userAgree" required />
        I agree: <input ng-show="user.agree" type="text" ng-model="user.agreeSign"
                  required /><br />
        <div ng-show="!user.agree || !user.agreeSign">Please agree and sign.</div>

        <button ng-click="reset()" ng-disabled="isUnchanged(user)">RESET</button>
        <button ng-click="update(user)"
                ng-disabled="form.$invalid || isUnchanged(user)">SAVE</button>
      </form>
    </div>
  </body>
</html>
```

script.js :

```
function Controller($scope) {
  $scope.master= {};

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

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

  $scope.isUnchanged = function(user) {
    return angular.equals(user, $scope.master);
  };

  $scope.reset();
}
```

Demo


Name:

E-mail:

Gender: ○male ○female

☐ I agree:

Please agree and sign.

Custom Validation

Angular provides basic implementation for most common html5 `input` types: (`text`, `number`, `url`, `email`, `radio`, `checkbox`), as well as some directives for validation (`required`, `pattern`, `minlength`, `maxlength`, `min`, `max`).

Defining your own validator can be done by defining your own directive which adds a custom validation function to the `ngModel` `controller`. To get a hold of the controller the directive specifies a dependency as shown in the example below. The validation can occur in two places:

- **Model to View update** - Whenever the bound model changes, all functions in `NgModelController#$formatters` array are pipe-lined, so that each of these functions has an opportunity to format the value and change validity state of the form control through `NgModelController#$setValidity`.
- **View to Model update** - In a similar way, whenever a user interacts with a control it calls `NgModelController#$setViewValue`. This in turn pipelines all functions in the `NgModelController#$parsers` array, so that each of these functions has an opportunity to convert the value and change validity state of the form control through `NgModelController#$setValidity`.

In the following example we create two directives.

- The first one is `integer` and it validates whether the input is a valid integer. For example `1.23` is an invalid value, since it contains a fraction. Note that we unshift the array instead of pushing. This is because we want to be first parser and consume the control string value, as we need to execute the validation function before a conversion to number occurs.
- The second directive is a `smart-float`. It parses both `1.2` and `1,2` into a valid float number `1.2`. Note that we can't use input type `number` here as HTML5 browsers would not allow the user to type what it would consider an invalid number such as `1,2`.

Source

index.html script.js Edit

index.html :

```
<!doctype html>
<html ng-app="form-example1">
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Controller">
      <form name="form" class="css-form" novalidate>
        <div>
          Size (integer 0 - 10):
          <input type="number" ng-model="size" name="size"
                 min="0" max="10" integer />{{size}}<br />
          <span ng-show="form.size.$error.integer">This is not valid integer!</span>
          <span ng-show="form.size.$error.min || form.size.$error.max">
            The value must be in range 0 to 10!</span>
        </div>

        <div>
          Length (float):
          <input type="text" ng-model="length" name="length" smart-float />
          {{length}}<br />
          <span ng-show="form.length.$error.float">
            This is not a valid float number!</span>
        </div>
      </form>
    </div>
  </body>
</html>
```

script.js :

```
var app = angular.module('form-example1', []);

var INTEGER_REGEXP = /^\-?\d*$/;
app.directive('integer', function() {
  return {
    require: 'ngModel',
    link: function(scope, elm, attrs, ctrl) {
      ctrl.$parsers.unshift(function(viewValue) {
        if (INTEGER_REGEXP.test(viewValue)) {
          // it is valid
          ctrl.$setValidity('integer', true);
          return viewValue;
        } else {
          // it is invalid, return undefined (no model update)
          ctrl.$setValidity('integer', false);
          return undefined;
        }
      });
    }
  };
});

var FLOAT_REGEXP = /^\-?\d+((\.|\,)\d+)?$/;
app.directive('smartFloat', function() {
  return {
    require: 'ngModel',
    link: function(scope, elm, attrs, ctrl) {
      ctrl.$parsers.unshift(function(viewValue) {
        if (FLOAT_REGEXP.test(viewValue)) {
          ctrl.$setValidity('float', true);
          return parseFloat(viewValue.replace(',', '.'));
        } else {
          ctrl.$setValidity('float', false);
          return undefined;
        }
      });
    }
  };
});
```

Demo

Size (integer 0 - 10):

Length (float):

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.

Source

index.html script.js Edit

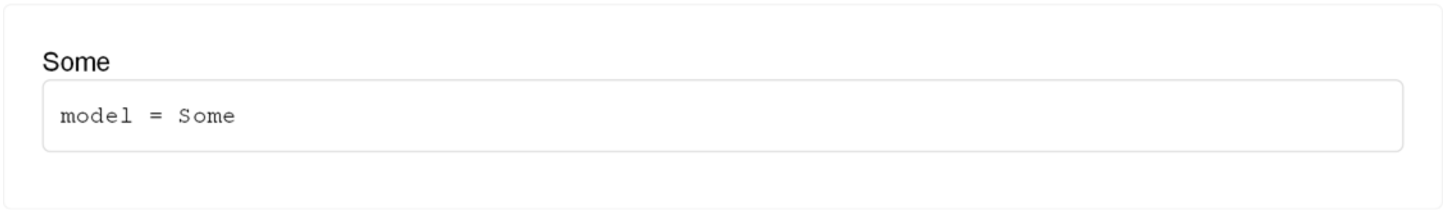
index.html :

```html
<!doctype html>
<html ng-app="form-example2">
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <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>
  </body>
</html>
```

script.js :

```javascript
angular.module('form-example2', []).directive('contenteditable', function() {
  return {
    require: 'ngModel',
    link: function(scope, elm, attrs, ctrl) {
      // view -> model
      elm.bind('blur', function() {
        scope.$apply(function() {
          ctrl.$setViewValue(elm.html());
        });
      });

      // model -> view
      ctrl.$render = function() {
        elm.html(ctrl.$viewValue);
      };

      // load init value from DOM
      ctrl.$setViewValue(elm.html());
    }
  };
});
```

Demo



Some

```
model = Some
```

I18n and L10n in AngularJS

What is i18n and l10n?

Internationalization, abbreviated i18n, is the process of developing products in such a way that they can be localized for languages and cultures easily. Localization, abbreviated 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.

What level of support for i18n/l10n is currently in Angular?

Currently, Angular supports i18n/l10n for datetime, number and currency filters.

Additionally, Angular supports localizable pluralization support provided by the `ngPluralize directive`.

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

For readers who want to jump straight into examples, we have a few web pages that showcase how to use Angular filters with various locale rule sets. You can find these examples either on Github or in the i18n/e2e folder of Angular development package.

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, 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 a an angular.js file that contains localization rules for german locale, you can do the following:

```
cat angular.js i18n/angular-locale_de-ge.js > angular_de-ge.js
```

When the application containing `angular_de-ge.js` script instead of the generic angular.js script starts, Angular is automatically pre-configured with localization rules for the german locale.

2. Including locale js script in index.html page

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-ge.html which will look something like this:

```
<html ng-app>
 <head>
….
```

```
    <script src="angular.js"></script>
    <script src="i18n/angular-locale_de-ge.js"></script>
 ….
  </head>
</html>
```

Comparison of the two approaches Both approaches described above requires you to prepare different index.html pages or js files for each locale that your app may be localized into. You also need to configure your server to serve the correct file that correspond to the desired locale.

However, the second approach (Including locale js script in index.html page) is likely to be slower because an extra script needs to be loaded.

"Gotchas"

Currency symbol "gotcha"

Angular's currency filter allows you to use the default currency symbol from the `locale service`, or you can provide the filter with a custom currency symbol. If your app will be used only in one locale, it is fine to rely on the default currency symbol. However, if you anticipate that viewers in other locales might use your app, you should provide your own currency symbol to make sure the actual value is understood.

For example, if you want to display account balance of 1000 dollars with the following binding containing currency filter: `{{ 1000 | currency }}`, and your app is currently in en-US locale. '$1000.00' will be shown. However, if someone in a different local (say, Japan) views your app, her browser will specify the locale as ja, and the balance of '¥1000.00' will be shown instead. This will really upset your client.

In this case, you need to override the default currency symbol by providing the currency filter with a currency symbol as a parameter when you configure the filter, for example, USD$1,000.00. This way, Angular will always show a balance of 'USD$1000' and disregard any locale changes.

Translation length "gotcha"

Keep in mind that translated strings/datetime formats can vary greatly in length. For example, `June 3, 1977` will be translated to Spanish as `3 de junio de 1977`. There are bound to be other more extreme cases. Hence, when internationalizing your apps, you need to apply CSS rules accordingly and do thorough testing to make sure UI components do not overlap.

Timezones

Keep in mind that Angular datetime filter uses the time zone settings of the browser. So the same application will show different time information depending on the time zone settings of the computer that the application is running on. Neither Javascript nor Angular currently supports displaying the date with a timezone specified by the developer.

Overview

This document describes the Internet Explorer (IE) idiosyncrasies when dealing with custom HTML attributes and tags. Read this document if you are planning on deploying your Angular application on IE v8.0 or earlier.

Short Version

To make your Angular application work on IE please make sure that:

1. You polyfill JSON.stringify if necessary (IE7 will need this). You can use JSON2 or JSON3 polyfills for this.
2. you **do not** use custom element tags such as `<ng:view>` (use the attribute version `<div ng-view>` instead), or
3. if you **do use** custom element tags, then you must take these steps to make IE happy:

```
<html xmlns:ng="http://angularjs.org">
  <head>
    <!--[if lte IE 8]>
      <script>
        document.createElement('ng-include');
        document.createElement('ng-pluralize');
        document.createElement('ng-view');

        // Optionally these for CSS
        document.createElement('ng:include');
        document.createElement('ng:pluralize');
        document.createElement('ng:view');
      </script>
    <![endif]-->
  </head>
  <body>
    ...
  </body>
</html>
```

The **important** parts are:

- `xmlns:ng` - *namespace* - you need one namespace for each custom tag you are planning on using.
- `document.createElement(yourTagName)` - *creation of custom tag names* - Since this is an issue only for older version of IE you need to load it conditionally. For each tag which does not have namespace and which is not defined in HTML you need to pre-declare it to make IE happy.

Long Version

IE has issues with element tag names which are not standard HTML tag names. These fall into two categories, and each category has its own fix.

- If the tag name starts with `my:` prefix than it is considered an XML namespace and must have corresponding namespace declaration on `<html xmlns:my="ignored">`
- If the tag has no `:` but it is not a standard HTML tag, then it must be pre-created using

`document.createElement('my-tag')`

- If you are planning on styling the custom tag with CSS selectors, then it must be pre-created using `document.createElement('my-tag')` regardless of XML namespace.

The Good News

The good news is that these restrictions only apply to element tag names, and not to element attribute names. So this requires no special handling in IE: `<div my-tag your:tag> </div>`.

What happens if I fail to do this?

Suppose you have HTML with unknown tag `mytag` (this could also be `my:tag` or `my-tag` with same result):

```
<html>
  <body>
    <mytag>some text</mytag>
  </body>
</html>
```

It should parse into the following DOM:

```
#document
  +- HTML
     +- BODY
        +- mytag
           +- #text: some text
```

The expected behavior is that the `BODY` element has a child element `mytag`, which in turn has the text `some text`.

But this is not what IE does (if the above fixes are not included):

```
#document
  +- HTML
     +- BODY
        +- mytag
        +- #text: some text
        +- /mytag
```

In IE, the behavior is that the `BODY` element has three children:

1. A self closing `mytag`. Example of self closing tag is `<br/>`. The trailing `/` is optional, but the `<br>` tag is not allowed to have any children, and browsers consider `<br>some text</br>` as three siblings not a `<br>` with `some text` as child.
2. A text node with `some text`. This should have been a child of `mytag` above, not a sibling.
3. A corrupt self closing `/mytag`. This is corrupt since element names are not allowed to have the `/` character. Furthermore this closing element should not be part of the DOM since it is only used to delineate the structure of the DOM.

CSS Styling of Custom Tag Names

To make CSS selectors work with custom elements, the custom element name must be pre-created with `document.createElement('my-tag')` regardless of XML namespace.

```html
<html xmlns:ng="needed for ng: namespace">
  <head>
    <!--[if lte IE 8]>
      <script>
        // needed to make ng-include parse properly
        document.createElement('ng-include');

        // needed to enable CSS reference
        document.createElement('ng:view');
      </script>
    <![endif]-->
    <style>
      ng\\:view {
        display: block;
        border: 1px solid red;
      }

      ng-include {
        display: block;
        border: 1px solid blue;
      }
    </style>
  </head>
  <body>
    <ng:view></ng:view>
    <ng-include></ng-include>
    ...
  </body>
</html>
```

Angular is pure client-side technology, written entirely in JavaScript. It works with the long-established technologies of the web (HTML, CSS, and JavaScript) to make the development of web apps easier and faster than ever before.

One important way that Angular simplifies web development is by increasing the level of abstraction between the developer and most low-level web app development tasks. Angular automatically takes care of many of these tasks, including:

- DOM Manipulation
- Setting Up Listeners and Notifiers
- Input Validation

Because Angular handles much of the work involved in these tasks, developers can concentrate more on application logic and less on repetitive, error-prone, lower-level coding.

At the same time that Angular simplifies the development of web apps, it brings relatively sophisticated techniques to the client-side, including:

- Separation of data, application logic, and presentation components
- Data Binding between data and presentation components
- Services (common web app operations, implemented as substitutable objects)
- Dependency Injection (used primarily for wiring together services)
- An extensible HTML compiler (written entirely in JavaScript)
- Ease of Testing

These techniques have been for the most part absent from the client-side for far too long.

Single-page / Round-trip Applications

You can use Angular to develop both single-page and round-trip apps, but Angular is designed primarily for developing single-page apps. Angular supports browser history, forward and back buttons, and bookmarking in single-page apps.

You normally wouldn't want to load Angular with every page change, as would be the case with using Angular in a round-trip app. However, it would make sense to do so if you were adding a subset of Angular's features (for example, templates to leverage angular's data-binding feature) to an existing round-trip app. You might follow this course of action if you were migrating an older app to a single-page Angular app.

What is a Module?

Most applications have a main method which instantiates, wires, and bootstraps 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 process is more declarative which is easier to understand
- In unit-testing there is no need to load all modules, which may aid in writing unit-tests.
- Additional modules can be loaded in scenario tests, which can override some of the configuration and help end-to-end test the application
- Third party code can be packaged as reusable modules.
- The modules can be loaded in any/parallel order (due to delayed nature of module execution).

The Basics

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

Important things to notice:

- `Module` API
- Notice the reference to the `myApp` module in the `<html ng-app="myApp">`, it is what bootstraps the app using your module.

Source

index.html   script.js   Edit

index.html :

```
<!doctype html>
<html ng-app="myApp">
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div>
      {{ 'World' | greet }}
    </div>
  </body>
</html>
```

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 + '!';
  };
});
```

Demo

Hello, World!

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 service module, for service declaration
- A directive module, for directive declaration
- A filter module, for filter declaration
- And an application level module which depends on the above modules, and which has initialization code.

The reason for this breakup is that in your tests, it is often necessary to ignore the initialization code, which tends to be difficult to test. By putting it into a separate module it can be easily ignored in tests. The tests can also be more focused by only loading the modules that are relevant to tests.

The above is only a suggestion, so feel free to tailor it to your needs.

Source

index.html script.js Edit

index.html :

```
<!doctype html>
<html ng-app="xmpl">
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="XmplController">
      {{ greeting }}!
    </div>
  </body>
</html>
```

script.js :

```
angular.module('xmpl.service', []).
  value('greeter', {
    salutation: 'Hello',
    localize: function(localization) {
      this.salutation = localization.salutation;
    },
    greet: function(name) {
      return this.salutation + ' ' + name + '!';
    }
  }).
  value('user', {
    load: function(name) {
      this.name = name;
    }
  });

angular.module('xmpl.directive', []);

angular.module('xmpl.filter', []);

angular.module('xmpl', ['xmpl.service', 'xmpl.directive', 'xmpl.filter']).
  run(function(greeter, user) {
    // This is effectively part of the main method initialization code
    greeter.localize({
      salutation: 'Bonjour'
    });
    user.load('World');
  })


// A Controller for your app
var XmplController = function($scope, greeter, user) {
  $scope.greeting = greeter.greet(user.name);
}
```

Demo

```
Bonjour World!!
```

Module Loading & Dependencies

A module is a collection of configuration and run blocks which get applied to the application during the bootstrap process. In its simplest form the module consist of 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 the 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 the 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', ...);
  });
```

The configuration blocks get applied in the order in which they are registered. The only exception to it are constant definitions, which are placed at the beginning of all configuration blocks.

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 service 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 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 or 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.

Unit Testing

In its simplest form a unit test is a way of instantiating a subset of the application in test and then applying a stimulus to it. It is important to realize that each module can only be loaded once per injector. Typically an app has only one injector. But in tests, each test has its own injector, which means that the modules are loaded multiple times per VM. Properly structured modules can help with unit testing, as in this example:

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:

```
describe('myApp', function() {
  // load the relevant application modules then load a special
  // test module which overrides the $window with a mock version,
  // so that calling window.alert() will not block the test
  // runner with a real alert box. This is an example of overriding
  // configuration information in tests.
  beforeEach(module('greetMod', function($provide) {
    $provide.value('$window', {
      alert: jasmine.createSpy('alert')
    });
  }));

  // The inject() will create the injector and inject the greet and
  // $window into the tests. The test need not concern itself with
  // wiring of the application, only with testing it.
  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 an inline module and inject methods.
  it('should alert using the alert service', function() {
    var alertSpy = jasmine.createSpy('alert');
    module(function($provide) {
      $provide.value('alert', alertSpy);
    });
    inject(function(greet) {
      greet('World');
      expect(alertSpy).toHaveBeenCalledWith('Hello World!');
    });
  });
});
```

What 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 isolate application components while providing access to shared model properties. A scope (prototypically) inherits properties from its parent scope.
- Scopes provide context against which expressions are evaluated. For example `{{username}}` expression is meaningless, unless it is evaluated against a specific scope which defines the `username` property.

Scope as Data-Model

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

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

Source

index.html script.js Edit

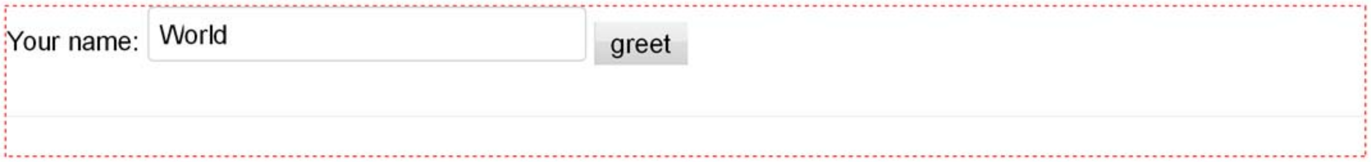
index.html :

```html
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="MyController">
      Your name:
        <input type="text" ng-model="username">
        <button ng-click='sayHello()'>greet</button>
      <hr>
      {{greeting}}
    </div>
  </body>
</html>
```

script.js :

```
function MyController($scope) {
  $scope.username = 'World';

  $scope.sayHello = function() {
    $scope.greeting = 'Hello ' + $scope.username + '!';
  };
}
```

Demo



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

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

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

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

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

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

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

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

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

Scope Hierarchies

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

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

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

This example illustrates scopes in application, and prototypical inheritance of properties.

Source

index.html style.css script.js Edit

index.html :

```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="EmployeeController">
      Manager: {{employee.name}} [ {{department}} ]<br>
      Reports:
        <ul>
          <li ng-repeat="employee in employee.reports">
            {{employee.name}} [ {{department}} ]
          </li>
        </ul>
      <hr>
      {{greeting}}
    </div>
  </body>
</html>
```

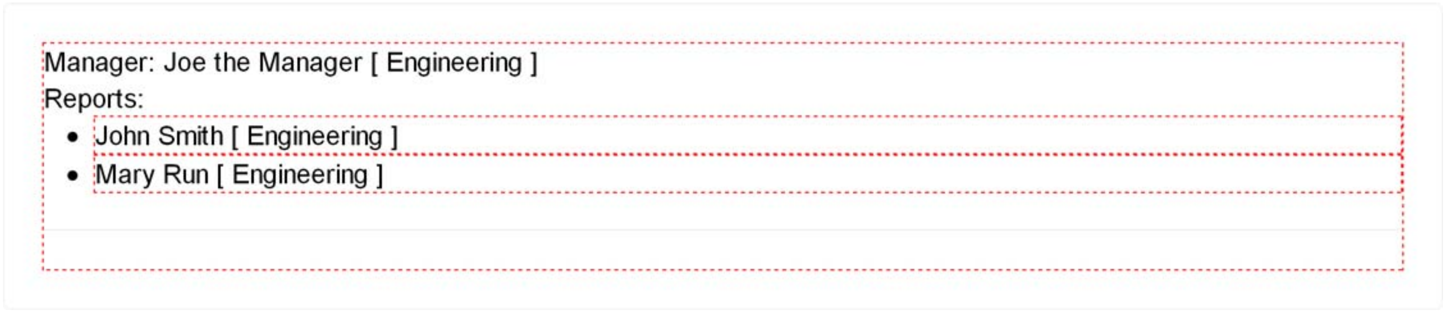
style.css :

```
/* remove .doc-example-live in jsfiddle */
.doc-example-live .ng-scope {
  border: 1px dashed red;
}
```

script.js :

```
function EmployeeController($scope) {
  $scope.department = 'Engineering';
  $scope.employee = {
    name: 'Joe the Manager',
    reports: [
      {name: 'John Smith'},
      {name: 'Mary Run'}
    ]
  };
}
```

Demo



Manager: Joe the Manager [Engineering]
Reports:

- John Smith [Engineering]
- Mary Run [Engineering]

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

Retrieving Scopes from the DOM.

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

To examine the scope in the debugger:

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

Scope Events Propagation

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

Source

index.html script.js Edit

index.html :

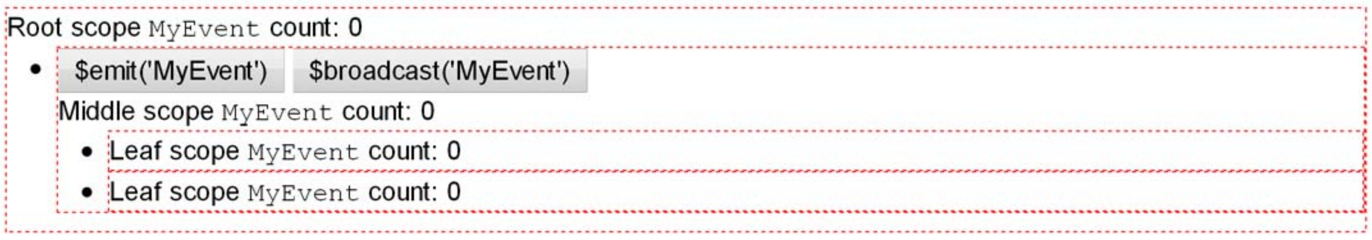
```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
```

```
      <script src="script.js"></script>
    </head>
    <body>
      <div ng-controller="EventController">
        Root scope <tt>MyEvent</tt> count: {{count}}
        <ul>
          <li ng-repeat="i in [1]" ng-controller="EventController">
            <button ng-click="$emit('MyEvent')">$emit('MyEvent')</button>
            <button ng-click="$broadcast('MyEvent')">$broadcast('MyEvent')</button>
            <br>
            Middle scope <tt>MyEvent</tt> count: {{count}}
            <ul>
              <li ng-repeat="item in [1, 2]" ng-controller="EventController">
                Leaf scope <tt>MyEvent</tt> count: {{count}}
              </li>
            </ul>
          </li>
        </ul>
      </div>
    </body>
  </html>
```

script.js :

```
function EventController($scope) {
  $scope.count = 0;
  $scope.$on('MyEvent', function() {
    $scope.count++;
  });
}
```

Demo


Root scope `MyEvent` count: 0

- $emit('MyEvent') $broadcast('MyEvent')

  Middle scope `MyEvent` count: 0
  - Leaf scope `MyEvent` count: 0
  - Leaf scope `MyEvent` count: 0

Scope Life Cycle

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

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

After evaluating the expression, the `$apply` method performs a `$digest`. In the $digest phase the scope examines all of the `$watch` expressions and compares them with the previous value. This dirty checking is done asynchronously. This means that assignment such as `$scope.username="angular"` will not immediately cause a `$watch` to be notified, instead the `$watch` notification is delayed until the `$digest` phase. This delay is desirable, since it coalesces multiple model updates into one `$watch` notification as well as it guarantees that during the `$watch` notification no other `$watches` are running. If a `$watch` changes the value of the model, it will force additional `$digest` cycle.

1. **Creation**

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

2. **Watcher registration**

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

3. **Model mutation**

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

4. **Mutation observation**

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

5. **Scope destruction**

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

Scopes and Directives

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

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

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

Directives that Create Scopes

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

Controllers and Scopes

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

- Controllers use scopes to expose controller methods to templates (see `ng-controller`).
- Controllers define methods (behavior) that can mutate the model (properties on the scope).

- Controllers may register `watches` on the model. These watches execute immediately after the controller behavior executes.

See the `ng-controller` for more information.

Scope `$watch` Performance Considerations

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

Dependency Injection

Dependency Injection (DI) is a software design pattern that deals with how code gets hold of its dependencies.

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.

DI in a nutshell

There are only three ways how an object or a function can get a hold of its dependencies:

1. The dependency can be created, typically using the `new` operator.
2. The dependency can be looked up by referring to a global variable.
3. The dependency can be passed in to where it is needed.

The first two option of creating or looking up dependencies are not optimal, because they hard code the dependency, making 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 the `SomeClass` is not concerned with locating the `greeter` dependency, it is simply handed the `greeter` at runtime.

This is desirable, but it puts the responsibility of getting hold of the dependency onto the code responsible for the construction of `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
angular.module('myModule', []).

  // Teach the injector how to build a 'greeter'
  // Notice that greeter itself is dependent on '$window'
  factory('greeter', function($window) {
    // This is a factory function, and is responsible for
    // creating the 'greet' service.
    return {
      greet: function(text) {
        $window.alert(text);
      }
    };
  });

// New injector is created from the module.
// (This is usually done automatically by angular bootstrap)
var injector = angular.injector(['myModule', 'ng']);

// Request any dependency from the injector
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 turn the dependency lookup responsibility to the injector by declaring the dependencies as in this example:

```
<!-- Given this HTML -->
<div ng-controller="MyController">
  <button ng-click="sayHello()">Hello</button>
</div>
```

```
// And this controller definition
function MyController($scope, greeter) {
  $scope.sayHello = function() {
    greeter.greet('Hello World');
  };
}

// The 'ng-controller' directive does this behind the scenes
injector.instantiate(MyController);
```

Notice that by having the `ng-controller` instantiate the class, it can satisfy all of the dependencies of the `MyController` without the controller ever knowing about the injector. This is the best outcome. The application code simply ask for the dependencies it needs, without having to deal with the injector. This setup does not break the Law of Demeter.

Dependency Annotation

How does the injector know what service needs to be injected?

The application developer needs to provide annotation information that the injector uses in order to resolve the dependencies. Throughout Angular certain API functions are invoked using the injector, as per the API documentation. The injector needs to know what services to inject into the function. Below are three equivalent ways of annotating your code with service name information. These can be used interchangeably as you see fit and are equivalent.

Inferring Dependencies

The simplest way to get hold of the dependencies, is to assume that the function parameter names are the names of the dependencies.

```
function MyController($scope, greeter) {
  ...
}
```

Given a function the injector can infer the names of the service 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.

While straightforward, this method will not work with JavaScript minifiers/obfuscators as they rename the method parameter names. This makes this way of annotating only useful for pretotyping, and demo applications.

`$inject` Annotation

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

```
var MyController = function(renamed$scope, renamedGreeter) {
  ...
}
MyController.$inject = ['$scope', 'greeter'];
```

Care must be taken that the `$inject` annotation is kept in sync with the actual arguments in the function declaration.

This method of annotation is useful for controller declarations since it assigns the annotation information with the function.

Inline Annotation

Sometimes using the `$inject` annotation style is not convenient such as when annotating directives.

For example:

```
someModule.factory('greeter', function($window) {
  ...;
});
```

Results in code bloat due to the need of temporary variable:

```
var greeterFactory = function(renamed$window) {
  ...;
};

greeterFactory.$inject = ['$window'];

someModule.factory('greeter', greeterFactory);
```

For this reason the third annotation style is provided as well.

```
someModule.factory('greeter', ['$window', function(renamed$window) {
  ...;
}]);
```

Keep in mind that all of the annotation styles are equivalent and can be used anywhere in Angular where injection is supported.

Where can I use DI?

DI is pervasive throughout Angular. It is typically used in controllers and factory methods.

DI in controllers

Controllers are classes which are responsible for application behavior. The recommended way of declaring controllers is:

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

Factory methods

Factory methods are responsible for creating most objects in Angular. Examples are directives, services, and filters. The factory methods are registered with the module, and the recommended way of declaring factories is:

```
angualar.module('myModule', []).
  config(['depProvider', function(depProvider){
    ...
  }]).
  factory('serviceId', ['depService', function(depService) {
    ...
  }]).
  directive('directiveName', ['depService', function(depService) {
    ...
  }]).
  filter('filterName', ['depService', function(depService) {
    ...
  }]).
  run(['depService', function(depService) {
    ...
  }]);
```

While Model-View-Controller (MVC) has acquired different shades of meaning over the years since it first appeared, Angular incorporates the basic principles behind the original MVC software design pattern into its way of building client-side web applications.

The MVC pattern summarized:

- Separate applications into distinct presentation, data, and logic components
- Encourage loose coupling between these components

Along with services and dependency injection, MVC makes angular applications better structured, easier to maintain and more testable.

The following topics explain how angular incorporates the MVC pattern into the angular way of developing web applications:

- Understanding the Model Component
- Understanding the Controller Component
- Understanding the View Component

Depending on the context of the discussion in the Angular documentation, the term *model* can refer to either a single object representing one entity (for example, a model called "phones" with its value being an array of phones) or the entire data model for the application (all entities).

In Angular, a model is any data that is reachable as a property of an angular Scope object. The name of the property is the model identifier and the value is any JavaScript object (including arrays and primitives).

The only requirement for a JavaScript object to be a model in Angular is that the object must be referenced by an Angular scope as a property of that scope object. This property reference can be created explicitly or implicitly.

You can create models by explicitly creating scope properties referencing JavaScript objects in the following ways:

- Make a direct property assignment to the scope object in JavaScript code; this most commonly occurs in controllers:

  ```
  function MyCtrl($scope) {
      // create property 'foo' on the MyCtrl's scope
      // and assign it an initial value 'bar'
      $scope.foo = 'bar';
  }
  ```

- Use an angular expression with an assignment operator in templates:

  ```
  <button ng-click="{{foos='ball'}}">Click me</button>
  ```

- Use `ngInit directive` in templates (for toy/example apps only, not recommended for real applications):

  ```
  <body ng-init=" foo = 'bar' ">
  ```

Angular creates models implicitly (by creating a scope property and assigning it a suitable value) when processing the following template constructs:

- Form input, select, textarea and other form elements:

  ```
  <input ng-model="query" value="fluffy cloud">
  ```

  The code above creates a model called "query" on the current scope with the value set to "fluffy cloud".

- An iterator declaration in `ngRepeater`:

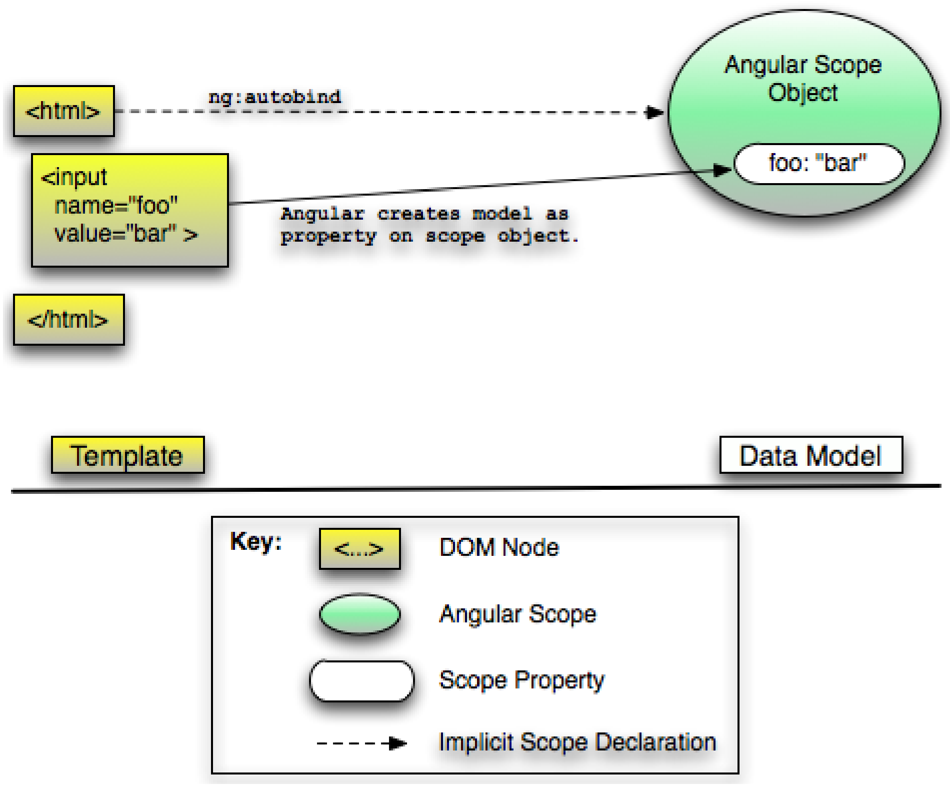
  ```
  <p ng-repeat="phone in phones"></p>
  ```

  The code above creates one child scope for each item in the "phones" array and creates a "phone" object (model) on each of these scopes with its value set to the value of "phone" in the array.

In Angular, a JavaScript object stops being a model when:

- No Angular scope contains a property that references the object.
- All Angular scopes that contain a property referencing the object become stale and eligible for garbage collection.

The following illustration shows a simple data model created implicitly from a simple template:



Related Topics

- About MVC in Angular
- Understanding the Controller Component
- Understanding the View Component

In Angular, a controller is a JavaScript function(type/class) that is used to augment instances of angular Scope, excluding the root scope. When you or Angular create a new child scope object via the `scope.$new` API , there is an option to pass in a controller as a method argument. This will tell Angular to associate the controller with the new scope and to augment its behavior.

Use controllers to:

- Set up the initial state of a scope object.
- Add behavior to the scope object.

Setting up the initial state of a scope object

Typically, when you create an application you need to set up an initial state for an Angular scope.

Angular applies (in the sense of JavaScript's `Function#apply`) the controller constructor function to a new Angular scope object, which sets up an initial scope state. This means that Angular never creates instances of the controller type (by invoking the `new` operator on the controller constructor). Constructors are always applied to an existing scope object.

You set up the initial state of a scope by creating model properties. For example:

function GreetingCtrl($scope) { $scope.greeting = 'Hola!'; }

The `GreetingCtrl` controller creates a `greeting` model which can be referred to in a template.

NOTE: Many of the examples in the documentation show the creation of functions in the global scope. This is only for demonstration purposes - in a real application you should use the `.controller` method of your Angular module for your application as follows:

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

myApp.controller('GreetingCtrl', ['$scope', function(scope) { scope.greeting = 'Hola!'; }]);

Note also that we use the array notation to explicitly specify the dependency of the controller on the `$scope` service provided by Angular.

Adding Behavior to a Scope Object

Behavior on an Angular scope object is in the form of scope method properties available to the template/view. This behavior interacts with and modifies the application model.

As discussed in the Model section of this guide, any objects (or primitives) assigned to the scope become model properties. Any functions assigned to the scope are available in the template/view, and can be invoked via angular expressions and `ng` event handler directives (e.g. `ngClick`).

Using Controllers Correctly

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

The most common way to keep controllers slim is by encapsulating work that doesn't belong to controllers into services and then using these services in controllers via dependency injection. This is discussed in the Dependency Injection Services sections of this guide.

Do not use controllers for:

- Any kind of DOM manipulation — Controllers should contain only business logic. DOM manipulation—the presentation logic of an application—is well known for being hard to test. Putting any presentation logic into controllers significantly

affects testability of the business logic. Angular offers databinding for automatic DOM manipulation. If you have to perform your own manual DOM manipulation, encapsulate the presentation logic in directives.
- Input formatting — Use angular form controls instead.
- Output filtering — Use angular filters instead.
- To run stateless or stateful code shared across controllers — Use angular services instead.
- To instantiate or manage the life-cycle of other components (for example, to create service instances).

Associating Controllers with Angular Scope Objects

You can associate controllers with scope objects explicitly via the `scope.$new` api or implicitly via the `ngController directive` or `$route service`.

Controller Constructor and Methods Example

To illustrate how the controller component works in angular, let's create a little app with the following components:

- A template with two buttons and a simple message
- A model consisting of a string named `spice`
- A controller with two functions that set the value of `spice`

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

A Spicy Controller Example

```
<body ng-controller="SpicyCtrl">
 <button ng-click="chiliSpicy()">Chili</button>
 <button ng-click="jalapenoSpicy()">Jalapeño</button>
 <p>The food is {{spice}} spicy!</p>
</body>

function SpicyCtrl($scope) {
 $scope.spice = 'very';
 $scope.chiliSpicy = function() {
   $scope.spice = 'chili';
 }
 $scope.jalapenoSpicy = function() {
  $scope.spice = 'jalapeño';
 }
}
```

Things to notice in the example above:

- The `ngController` directive is used to (implicitly) create a scope for our template, and the scope is augmented (managed) by the `SpicyCtrl` controller.
- `SpicyCtrl` is just a plain JavaScript function. As an (optional) naming convention the name starts with capital letter and ends with "Ctrl" or "Controller".
- Assigning a property to `$scope` creates or updates the model.
- Controller methods can be created through direct assignment to scope (the `chiliSpicy` method)
- Both controller methods are available in the template (for the `body` element and and its children).
- NB: Previous versions of Angular (pre 1.0 RC) allowed you to use `this` interchangeably with the $scope method, but this is no longer the case. Inside of methods defined on the scope `this` and $scope are interchangeable (angular sets

`this` to $scope), but not otherwise inside your controller constructor.
- NB: Previous versions of Angular (pre 1.0 RC) added prototype methods into the scope automatically, but this is no longer the case; all methods need to be added manually to the scope.

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

Controller Method Arguments Example

```
<body ng-controller="SpicyCtrl">
 <input ng-model="customSpice" value="wasabi">
 <button ng-click="spicy('chili')">Chili</button>
 <button ng-click="spicy(customSpice)">Custom spice</button>
 <p>The food is {{spice}} spicy!</p>
</body>

function SpicyCtrl($scope) {
 $scope.spice = 'very';
 $scope.spicy = function(spice) {
   $scope.spice = spice;
 }
}
```

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

Controller Inheritance Example

Controller inheritance in Angular is based on `Scope` inheritance. Let's have a look at an example:

```
<body ng-controller="MainCtrl">
 <p>Good {{timeOfDay}}, {{name}}!</p>
 <div ng-controller="ChildCtrl">
   <p>Good {{timeOfDay}}, {{name}}!</p>
   <p ng-controller="BabyCtrl">Good {{timeOfDay}}, {{name}}!</p>
</body>

function MainCtrl($scope) {
 $scope.timeOfDay = 'morning';
 $scope.name = 'Nikki';
}

function ChildCtrl($scope) {
 $scope.name = 'Mattie';
}

function BabyCtrl($scope) {
 $scope.timeOfDay = 'evening';
 $scope.name = 'Gingerbreak Baby';
}
```

Notice how we nested three `ngController` directives in our template. This template construct will result in 4 scopes being created for our view:

- The root scope
- The `MainCtrl` scope, which contains `timeOfDay` and `name` models
- The `ChildCtrl` scope, which shadows the `name` model from the previous scope and inherits the `timeOfDay` model
- The `BabyCtrl` scope, which shadows both the `timeOfDay` model defined in `MainCtrl` and `name` model defined in the ChildCtrl

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

Note: Standard prototypical inheritance between two controllers doesn't work as one might expect, because as we mentioned earlier, controllers are not instantiated directly by Angular, but rather are applied to the scope object.

Testing Controllers

Although there are many ways to test a controller, one of the best conventions, shown below, involves injecting the `$rootScope` and `$controller`

Controller Function:

```
function myController($scope) {
   $scope.spices = [{"name":"pasilla", "spiciness":"mild"},
                  {"name":"jalapeno", "spiceiness":"hot hot hot!"},
                  {"name":"habanero", "spiceness":"LAVA HOT!!"}];

   $scope.spice = "habanero";
}
```

Controller Test:

```
describe('myController function', function() {

  describe('myController', function() {
    var scope;

    beforeEach(inject(function($rootScope, $controller) {
      scope = $rootScope.$new();
      var ctrl = $controller(myController, {$scope: scope});
    }));

    it('should create "spices" model with 3 spices', function() {
      expect(scope.spices.length).toBe(3);
    });

    it('should set the default value of spice', function() {
      expect(scope.spice).toBe('habanero');
    });
  });
});
```

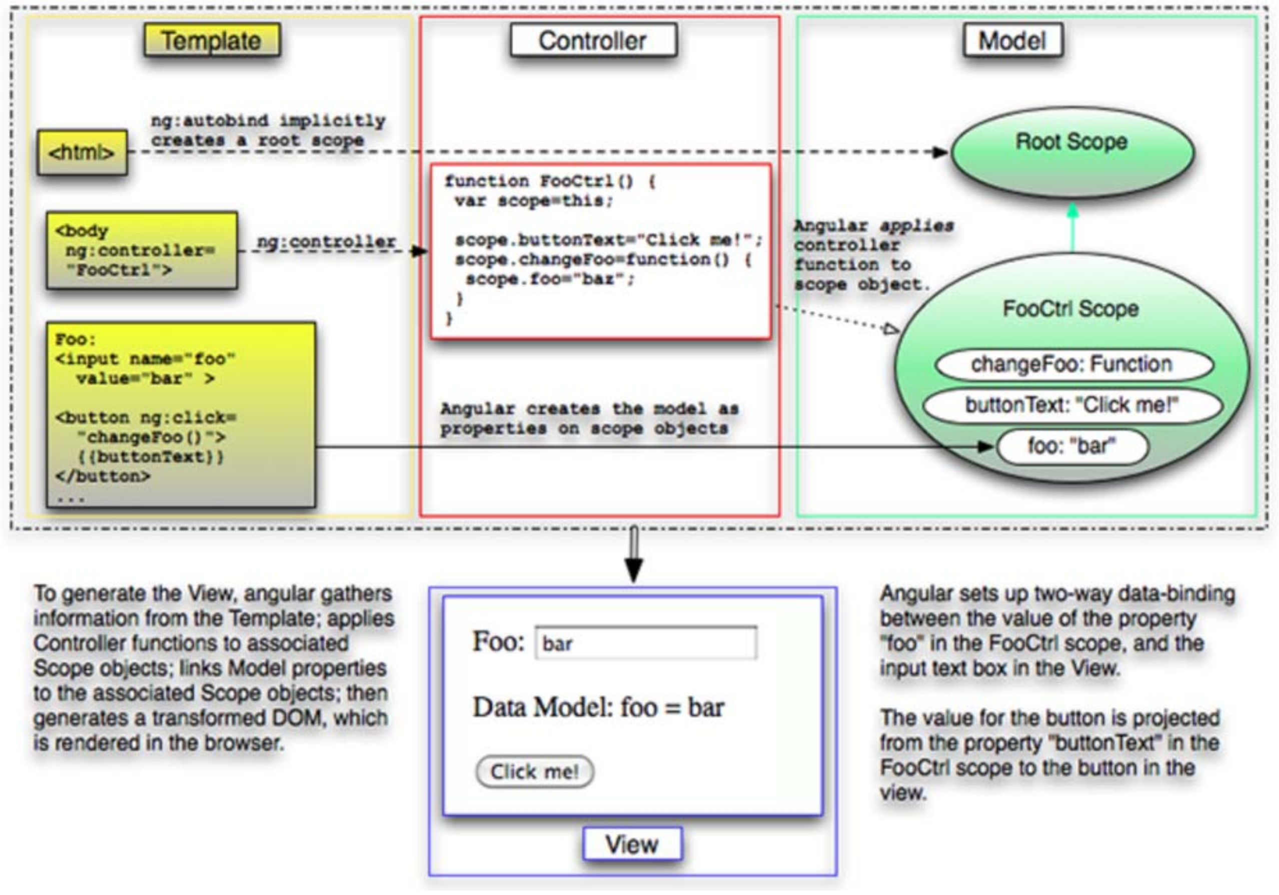
If you need to test a nested controller you need to create the same scope hierarchy in your test that exists in the DOM.

```
describe('state', function() {
    var mainScope, childScope, babyScope;

    beforeEach(inject(function($rootScope, $controller) {
        mainScope = $rootScope.$new();
        var mainCtrl = $controller(MainCtrl, {$scope: mainScope});
        childScope = mainScope.$new();
        var childCtrl = $controller(ChildCtrl, {$scope: childScope});
        babyScope = childCtrl.$new();
        var babyCtrl = $controller(BabyCtrl, {$scope: babyScope});
    }));

    it('should have over and selected', function() {
        expect(mainScope.timeOfDay).toBe('morning');
        expect(mainScope.name).toBe('Nikki');
        expect(childScope.timeOfDay).toBe('morning');
        expect(childScope.name).toBe('Mattie');
        expect(babyScope.timeOfDay).toBe('evening');
        expect(babyScope.name).toBe('Gingerbreak Baby');
    });
});
```

Related Topics

- About MVC in Angular
- Understanding the Model Component
- Understanding the View Component

In Angular, the view is the DOM loaded and rendered in the browser, after Angular has transformed the DOM based on information in the template, controller and model.


Template
Controller
Model
ng:autobind implicitly creates a root scope
<html>
Root Scope
<body ng:controller="FooCtrl">
ng:controller
function FooCtrl() {
 var scope=this;
 scope.buttonText="Click me!";
 scope.changeFoo=function() {
  scope.foo="baz";
 }
}
Angular applies controller function to scope object.
FooCtrl Scope
changeFoo: Function
buttonText: "Click me!"
foo: "bar"
Foo:
<input name="foo" value="bar" >
<button ng:click="changeFoo()">{{buttonText}}</button>
...
Angular creates the model as properties on scope objects
To generate the View, angular gathers information from the Template; applies Controller functions to associated Scope objects; links Model properties to the associated Scope objects; then generates a transformed DOM, which is rendered in the browser.
Foo: bar
Data Model: foo = bar
Click me!
View
Angular sets up two-way data-binding between the value of the property "foo" in the FooCtrl scope, and the input text box in the View.
The value for the button is projected from the property "buttonText" in the FooCtrl scope to the button in the view.

In the Angular implementation of MVC, the view has knowledge of both the model and the controller. The view knows about the model where two-way data-binding occurs. The view has knowledge of the controller through Angular directives, such as `ngController` and `ngView`, and through bindings of this form: `{{someControllerFunction()}}`. In these ways, the view can call functions in an associated controller function.

Related Topics

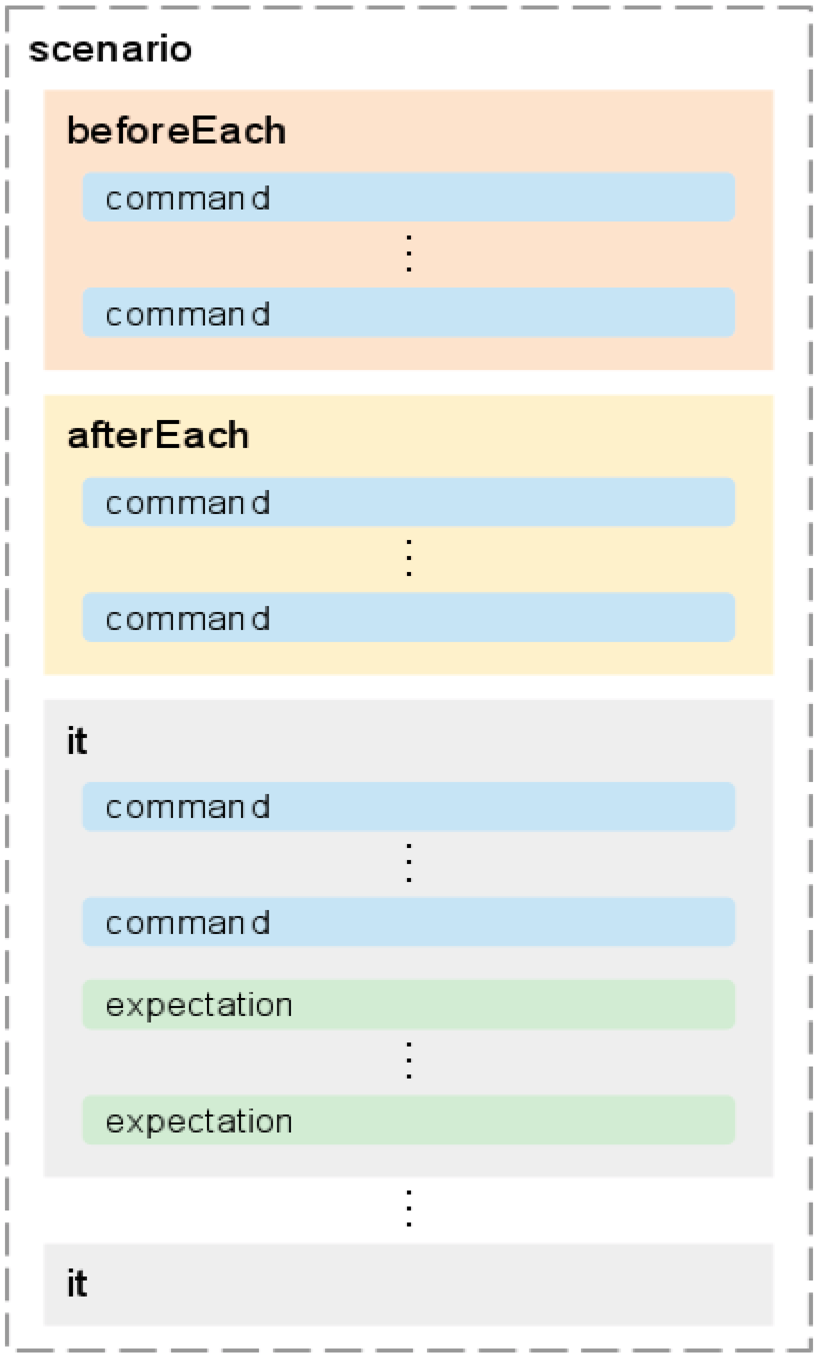
- About MVC in Angular
- Understanding the Model Component
- Understanding the Controller Component

As applications grow in size and complexity, it becomes unrealistic to rely on manual testing to verify the correctness of new features, catch bugs and notice regressions.

To solve this problem, we have built an Angular Scenario Runner which simulates user interactions that will help you verify the health of your Angular application.

Overview

You will write scenario tests in JavaScript, which describe how your application should behave, given a certain interaction in a specific state. A scenario is comprised of one or more `it` blocks (you can think of these as the requirements of your application), which in turn are made of **commands** and **expectations**. Commands tell the Runner to do something with the application (such as navigate to a page or click on a button), and expectations tell the Runner to assert something about the state (such as the value of a field or the current URL). If any expectation fails, the runner marks the `it` as "failed" and continues on to the next one. Scenarios may also have **beforeEach** and **afterEach** blocks, which will be run before (or after) each `it` block, regardless of whether they pass or fail.



In addition to the above elements, scenarios may also contain helper functions to avoid duplicating code in the `it` blocks.

Here is an example of a simple scenario:

```
describe('Buzz Client', function() {
it('should filter results', function() {
  input('user').enter('jacksparrow');
  element(':button').click();
  expect(repeater('ul li').count()).toEqual(10);
  input('filterText').enter('Bees');
  expect(repeater('ul li').count()).toEqual(1);
});
});
```

This scenario describes the requirements of a Buzz Client, specifically, that it should be able to filter the stream of the user. It starts by entering a value in the 'user' input field, clicking the only button on the page, and then it verifies that there are 10 items listed. It then enters 'Bees' in the 'filterText' input field and verifies that the list is reduced to a single item.

The API section below lists the available commands and expectations for the Runner.

API

Source: https://github.com/angular/angular.js/blob/master/src/ngScenario/dsl.js

pause()

Pauses the execution of the tests until you call `resume()` in the console (or click the resume link in the Runner UI).

sleep(seconds)

Pauses the execution of the tests for the specified number of `seconds`.

browser().navigateTo(url)

Loads the `url` into the test frame.

browser().navigateTo(url, fn)

Loads the URL returned by `fn` into the testing frame. The given `url` is only used for the test output. Use this when the destination URL is dynamic (that is, the destination is unknown when you write the test).

browser().reload()

Refreshes the currently loaded page in the test frame.

browser().window().href()

Returns the window.location.href of the currently loaded page in the test frame.

browser().window().path()

Returns the window.location.pathname of the currently loaded page in the test frame.

browser().window().search()

Returns the window.location.search of the currently loaded page in the test frame.

browser().window().hash()

Returns the window.location.hash (without #) of the currently loaded page in the test frame.

browser().location().url()

Returns the `$location.url()` of the currently loaded page in the test frame.

browser().location().path()

Returns the `$location.path()` of the currently loaded page in the test frame.

browser().location().search()

Returns the `$location.search()` of the currently loaded page in the test frame.

browser().location().hash()

Returns the `$location.hash()` of the currently loaded page in the test frame.

expect(future).{matcher}

Asserts the value of the given `future` satisfies the `matcher`. All API statements return a `future` object, which get a `value` assigned after they are executed. Matchers are defined using `angular.scenario.matcher`, and they use the value of futures to run the expectation. For example:
`expect(browser().location().href()).toEqual('http://www.google.com')`

expect(future).not().{matcher}

Asserts the value of the given `future` satisfies the negation of the `matcher`.

using(selector, label)

Scopes the next DSL element selection.

binding(name)

Returns the value of the first binding matching the given `name`.

input(name).enter(value)

Enters the given `value` in the text field with the given `name`.

input(name).check()

Checks/unchecks the checkbox with the given `name`.

input(name).select(value)

Selects the given `value` in the radio button with the given `name`.

input(name).val()

Returns the current value of an input field with the given `name`.

repeater(selector, label).count()

Returns the number of rows in the repeater matching the given jQuery `selector`. The `label` is used for test output.

repeater(selector, label).row(index)

Returns an array with the bindings in the row at the given `index` in the repeater matching the given jQuery `selector`. The `label` is used for test output.

repeater(selector, label).column(binding)

Returns an array with the values in the column with the given `binding` in the repeater matching the given jQuery `selector`. The `label` is used for test output.

select(name).option(value)

Picks the option with the given `value` on the select with the given `name`.

select(name).option(value1, value2...)

Picks the options with the given `values` on the multi select with the given `name`.

element(selector, label).count()

Returns the number of elements that match the given jQuery `selector`. The `label` is used for test output.

element(selector, label).click()

Clicks on the element matching the given jQuery `selector`. The `label` is used for test output.

element(selector, label).query(fn)

Executes the function `fn(selectedElements, done)`, where selectedElements are the elements that match the given jQuery `selector` and `done` is a function that is called at the end of the `fn` function. The `label` is used for test output.

element(selector, label).{method}()

Returns the result of calling `method` on the element matching the given jQuery `selector`, where `method` can be any of the following jQuery methods: `val`, `text`, `html`, `height`, `innerHeight`, `outerHeight`, `width`, `innerWidth`, `outerWidth`, `position`, `scrollLeft`, `scrollTop`, `offset`. The `label` is used for test output.

element(selector, label).{method}(value)

Executes the `method` passing in `value` on the element matching the given jQuery `selector`, where `method` can be any of the following jQuery methods: `val`, `text`, `html`, `height`, `innerHeight`, `outerHeight`, `width`, `innerWidth`, `outerWidth`, `position`, `scrollLeft`, `scrollTop`, `offset`. The `label` is used for test output.

element(selector, label).{method}(key)

Returns the result of calling `method` passing in `key` on the element matching the given jQuery `selector`, where `method` can be any of the following jQuery methods: `attr`, `prop`, `css`. The `label` is used for test output.

element(selector, label).{method}(key, value)

Executes the `method` passing in `key` and `value` on the element matching the given jQuery `selector`, where `method` can be any of the following jQuery methods: `attr`, `prop`, `css`. The `label` is used for test output.

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

An Angular template is the declarative specification that, along with information from the model and controller, becomes the rendered view that a user sees in the browser. It is the static DOM, containing HTML, CSS, and angular-specific elements and angular-specific element attributes. The Angular elements and attributes direct angular to add behavior and transform the template DOM into the dynamic view DOM.

These are the types of Angular elements and element attributes you can use in a template:

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

Note: In addition to declaring the elements above in templates, you can also access these elements in JavaScript code.

The following code snippet shows a simple Angular template made up of standard HTML tags along with Angular directives and curly-brace bindings with expressions:

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

In a simple single-page app, the template consists of HTML, CSS, and angular directives contained in just one HTML file (usually `index.html`). In a more complex app, you can display multiple views within one main page using "partials", which are segments of template located in separate HTML files. You "include" the partials in the main page using the `$route` service in conjunction with the `ngView` directive. An example of this technique is shown in the angular tutorial, in steps seven and eight.

Related Topics

- Angular Filters
- Angular Forms

Related API

- API Reference

Angular sets these CSS classes. It is up to your application to provide useful styling.

CSS classes used by angular

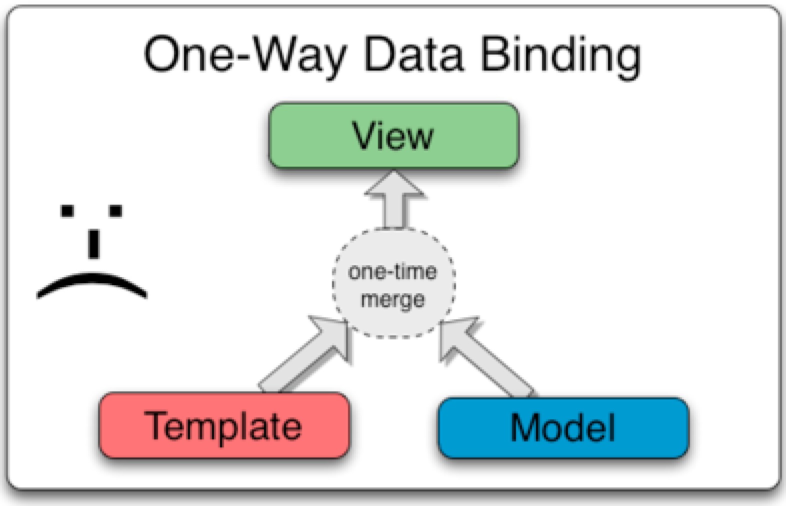
- `ng-invalid`, `ng-valid`
  - **Usage:** angular applies this class to an input widget element if that element's input does not pass validation. (see `input` directive).
- `ng-pristine`, `ng-dirty`
  - **Usage:** angular `input` directive applies `ng-pristine` class to a new input widget element which did not have user interaction. Once the user interacts with the input widget the class is changed to `ng-dirty`.

Related Topics

- Angular Templates
- Angular Forms

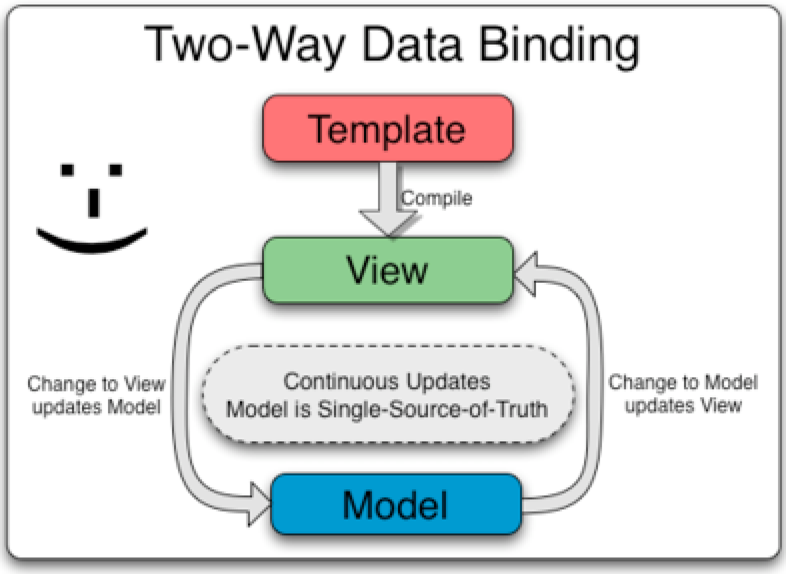
Data-binding in Angular web apps is the automatic syncronization of data between the model and view components. The way that Angular implements data-binding lets you treat the model as the single-source-of-truth in your application. The view is a projection of the model at all times. When the model changes,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

The way Angular templates works is different, as illustrated in the diagram. They are different because first the template (which is the uncompiled HTML along with any additional markup or directives) is compiled on the browser, and second, the compilation step produces a live view. We say live because any changes to the view are immediately reflected in the model, and any changes in the model are propagated to the view. This makes the model always 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.

Related Topics

- Angular Scopes
- Angular Templates

Angular filters format data for display to the user. In addition to formatting data, filters can also modify the DOM. This allows filters to handle tasks such as conditionally applying CSS styles to filtered output.

For example, you might have a data object that needs to be formatted according to the locale before displaying it to the user. You can pass expressions through a chain of filters like this:

```
    name | uppercase
```

The expression evaluator simply passes the value of name to `uppercase filter`.

Related Topics

- Using Angular Filters
- Creating Angular Filters

Related API

- `Angular Filter API`

Writing your own filter is very easy: just register a new filter (injectable) factory function with your module. This factory function should return a new filter function which takes the input value as the first argument. Any filter arguments are passed in as additional arguments to the filter function.

The following sample filter reverses a text string. In addition, it conditionally makes the text upper-case and assigns color.

Source

index.html script.js End to end test Edit

index.html :

```
<!doctype html>
<html ng-app="MyReverseModule">
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="Ctrl">
      <input ng-model="greeting" type="greeting"><br>
      No filter: {{greeting}}<br>
      Reverse: {{greeting|reverse}}<br>
      Reverse + uppercase: {{greeting|reverse:true}}<br>
    </div>
  </body>
</html>
```

script.js :

```
angular.module('MyReverseModule', []).
  filter('reverse', function() {
    return function(input, uppercase) {
      var out = "";
      for (var i = 0; i < input.length; i++) {
        out = input.charAt(i) + out;
      }
      // conditional based on optional argument
      if (uppercase) {
        out = out.toUpperCase();
      }
      return out;
    }
  });

function Ctrl($scope) {
  $scope.greeting = 'hello';
}
```

End to end test :

```
it('should reverse greeting', function() {
  expect(binding('greeting|reverse')).toEqual('olleh');
  input('greeting').enter('ABC');
  expect(binding('greeting|reverse')).toEqual('CBA');
});
```

Demo

hello

No filter: hello
Reverse: olleh
Reverse + uppercase: OLLEH

Related Topics

- Understanding Angular Filters
- Angular HTML Compiler

Related API

- `Angular Filter API`

Filters can be part of any `api/ng.$rootScope.Scope` evaluation but are typically used to format expressions in bindings in your templates:

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

Filters typically transform the data to a new data type, formatting the data in the process. Filters can also be chained, and can take optional arguments.

You can chain filters using this syntax:

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

You can also pass colon-delimited arguments to filters, for example, to display the number 123 with 2 decimal points:

```
123 | number:2
```

Here are some examples that show values before and after applying different filters to an expression in a binding:

- No filter: `{{1234.5678}}` => `1234.5678`
- Number filter: `{{1234.5678|number}}` => `1,234.57`. Notice the "," and rounding to two significant digits.
- Filter with arguments: `{{1234.5678|number:5}}` => `1,234.56780`. Filters can take optional arguments, separated by colons in a binding. For example, the "number" filter takes a number argument that specifies how many digits to display to the right of the decimal point.

Related Topics

- Understanding Angular Filters
- Creating Angular Filters

Related API

- `Angular Filter API`

Services are a feature that Angular brings to client-side web apps from the server side, where services have been commonly used for a long time. Services in Angular apps are substitutable objects that are wired together using dependency injection (DI).

Related Topics

- Understanding Angular Services
- Creating Angular Services
- Managing Service Dependencies
- Injecting Services Into Controllers
- Testing Angular Services

Related API

- Angular Service API

What does it do?

The `$location` service parses the URL in the browser address bar (based on the window.location) and makes the URL available to your application. Changes to the URL in the address bar are reflected into $location service and changes to $location are reflected into the browser address bar.

The $location service:

- Exposes the current URL in the browser address bar, so you can
  - Watch and observe the URL.
  - Change the URL.
- Synchronizes the URL with the browser when the user
  - Changes the address bar.
  - Clicks the back or forward button (or clicks a History link).
  - Clicks on a link.
- Represents the URL object as a set of methods (protocol, host, port, path, search, hash).

Comparing $location to window.location

	window.location	**$location service**
purpose	allow read/write access to the current browser location	same
API	exposes "raw" object with properties that can be directly modified	exposes jQuery-style getters and setters
integration with angular application life-cycle	none	knows about all internal life-cycle phases, integrates with $watch, ...
seamless integration with HTML5 API	no	yes (with a fallback for legacy browsers)
aware of docroot/context from which the application is loaded	no - window.location.path returns "/docroot/actual/path"	yes - $location.path() returns "/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}
  `true` - see HTML5 mode
  `false` - see Hashbang mode
  default: `false`
- **hashPrefix(prefix)**: {string}
  prefix used for Hashbang URLs (used in Hashbang mode or in legacy browser 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});
```

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.

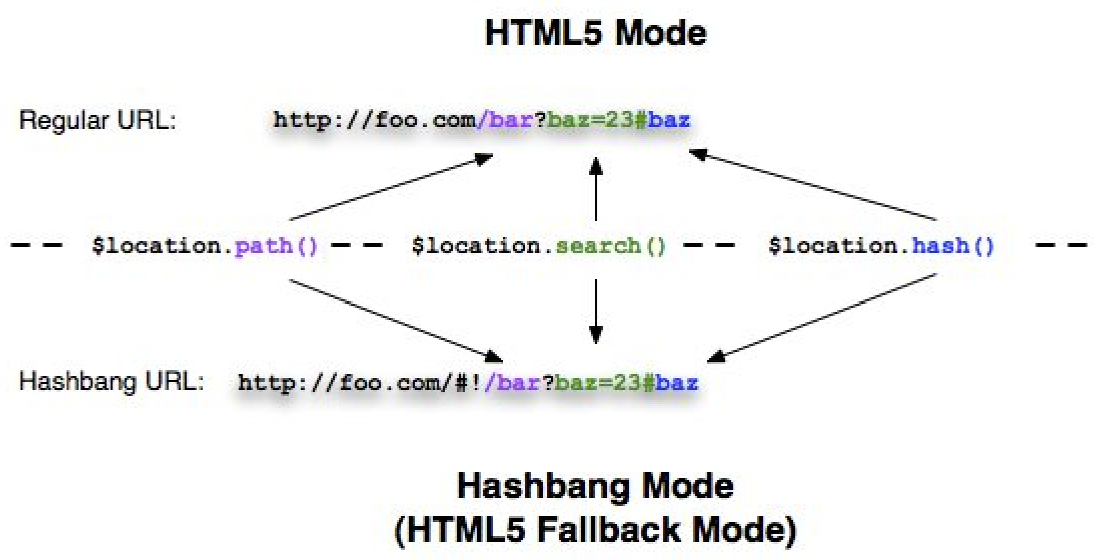
Setters and character encoding

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 modern browser, hashbang URLs in old browser
<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.

Example

```
it('should show example', inject(
  function($locationProvider) {
    $locationProvider.html5Mode(false);
    $locationProvider.hashPrefix = '!';
  },
  function($location) {
    // open http://host.com/base/index.html#!/a
    $location.absUrl() == 'http://host.com/base/index.html#!/a'
    $location.path() == '/a'

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

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

    $location.path('/new').search('x=y');
    $location.absUrl() == 'http://host.com/base/index.html#!/new?x=y'
  }
));
```

Crawling your app

To allow indexing of your AJAX application, you have to add special meta tag in the head section of your document:

```
<meta name="fragment" content="!" />
```

This will cause crawler bot to request links with `_escaped_fragment_` param so that your server can recognize the crawler and serve a HTML snapshots. For more information about this technique, see Making AJAX Applications Crawlable.

HTML5 mode

In HTML5 mode, the `$location` service getters and setters interact with the browser URL address through the HTML5 history API, which allows for use of regular URL path and search segments, instead of their hashbang equivalents. If the HTML5 History API is not supported by a browser, the `$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

Example

```
it('should show example', inject(
  function($locationProvider) {
    $locationProvider.html5Mode(true);
    $locationProvider.hashPrefix = '!';
  },
  function($location) {
    // in browser with HTML5 history support:
    // open http://host.com/#!/a -> rewrite to http://host.com/a
    // (replacing the http://host.com/#!/a history record)
    $location.path() == '/a'
```

```

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

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

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

      // in browser without html5 history support:
      // open http://host.com/new?x=y -> redirect to http://host.com/#!/new?x=y
      // (again replacing the http://host.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://host.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 Hasbang 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 need different links in different browsers, but 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 when base is defined
  Example: `<a href="/not-my-base/link">link</a>`

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)

Crawling your app

If you want your AJAX application to be indexed by web crawlers, you will need to add the following meta tag to the HEAD section of your document:

```
<meta name="fragment" content="!" />
```

This statement causes a crawler to request links with an empty `_escaped_fragment_` parameter so that your server can recognize the crawler and serve it HTML snapshots. For more information about this technique, see Making AJAX Applications Crawlable.

Relative links

Be sure to check all relative links, images, scripts etc. You must either specify the url base in the head of your main html file (`<base href="/my-base">`) or you must use absolute urls (starting with `/`) everywhere because relative urls will be resolved to absolute urls using the initial absolute url of the document, which is often different from the root of the application.

Running Angular apps with the History API enabled from document root is strongly encouraged as it takes care of all relative link issues.

Sending links among different browsers

Because of rewriting capability in HTML5 mode, your users will be able to open regular url links in legacy browsers and hashbang links in modern browser:

- Modern browser will rewrite hashbang URLs to regular URLs.
- Older browsers will redirect regular URLs to hashbang URLs.

Example

Here you can see two `$location` instances, both in **Html5 mode**, but on different browsers, so that you can see the differences. These `$location` services are connected to a fake browsers. Each input represents address bar of the browser.

Note that when you type hashbang url into first browser (or vice versa) it doesn't rewrite / redirect to regular / hashbang url, as this conversion happens only during parsing the initial URL = on page reload.

In this examples we use `<base href="/base/index.html" />`

Source

index.html script.js Edit

index.html :

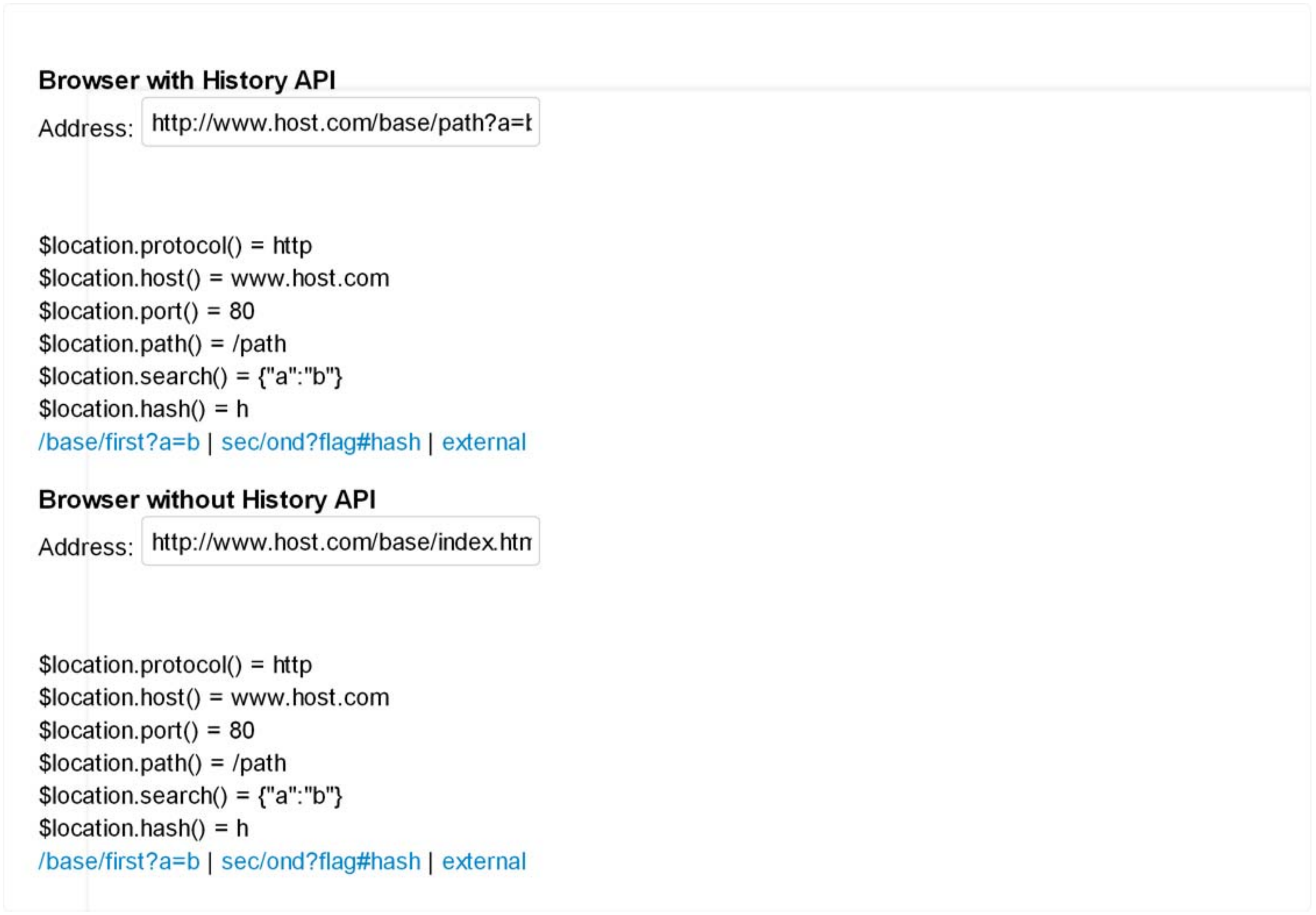
```
<!doctype html>
<html ng-app>
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-non-bindable class="html5-hashbang-example">
      <div id="html5-mode" ng-controller="Html5Cntl">
        <h4>Browser with History API</h4>
        <div ng-address-bar browser="html5"></div><br><br>
        $location.protocol() = {{$location.protocol()}}<br>
        $location.host() = {{$location.host()}}<br>
        $location.port() = {{$location.port()}}<br>
        $location.path() = {{$location.path()}}<br>
        $location.search() = {{$location.search()}}<br>
        $location.hash() = {{$location.hash()}}<br>
        <a href="http://www.host.com/base/first?a=b">/base/first?a=b</a> |
        <a href="http://www.host.com/base/sec/ond?flag#hash">sec/ond?flag#hash</a> |
        <a href="/other-base/another?search">external</a>
      </div>

      <div id="hashbang-mode" ng-controller="HashbangCntl">
        <h4>Browser without History API</h4>
        <div ng-address-bar browser="hashbang"></div><br><br>
        $location.protocol() = {{$location.protocol()}}<br>
        $location.host() = {{$location.host()}}<br>
        $location.port() = {{$location.port()}}<br>
        $location.path() = {{$location.path()}}<br>
        $location.search() = {{$location.search()}}<br>
        $location.hash() = {{$location.hash()}}<br>
        <a href="http://www.host.com/base/first?a=b">/base/first?a=b</a> |
        <a href="http://www.host.com/base/sec/ond?flag#hash">sec/ond?flag#hash</a> |
        <a href="/other-base/another?search">external</a>
      </div>
    </div>
  </body>
</html>
```

script.js :

```
function FakeBrowser(initUrl, baseHref) {
  this.onUrlChange = function(fn) {
    this.urlChange = fn;
  };

  this.url = function() {
    return initUrl;
  };

  this.defer = function(fn, delay) {
    setTimeout(function() { fn(); }, delay || 0);
  };

  this.baseHref = function() {
    return baseHref;
  };

  this.notifyWhenOutstandingRequests = angular.noop;
}

var browsers = {
  html5: new FakeBrowser('http://www.host.com/base/path?a=b#h', '/base/index.html'),
  hashbang: new FakeBrowser('http://www.host.com/base/index.html#!/path?a=b#h', '/base
/index.html')
};

```

```
function Html5Cntl($scope, $location) {
  $scope.$location = $location;
}

function HashbangCntl($scope, $location) {
  $scope.$location = $location;
}

function initEnv(name) {
  var root = angular.element(document.getElementById(name + '-mode'));
  angular.bootstrap(root, [function($compileProvider, $locationProvider, $provide){
    $locationProvider.html5Mode(true).hashPrefix('!');

    $provide.value('$browser', browsers[name]);
    $provide.value('$document', root);
    $provide.value('$sniffer', {history: name == 'html5'});

    $compileProvider.directive('ngAddressBar', function() {
      return function(scope, elm, attrs) {
        var browser = browsers[attrs.browser],
            input = angular.element('<input type="text">').val(browser.url()),
            delay;

        input.bind('keypress keyup keydown', function() {
          if (!delay) {
            delay = setTimeout(fireUrlChange, 250);
          }
        });

        browser.url = function(url) {
          return input.val(url);
        };

        elm.append('Address: ').append(input);

        function fireUrlChange() {
          delay = null;
          browser.urlChange(input.val());
        }
      };
    });
  }]);
  root.bind('click', function(e) {
    e.stopPropagation();
  });
}

initEnv('html5');
initEnv('hashbang');
```

Demo


Browser with History API

Address: http://www.host.com/base/path?a=b

$location.protocol() = http
$location.host() = www.host.com
$location.port() = 80
$location.path() = /path
$location.search() = {"a":"b"}
$location.hash() = h
/base/first?a=b | sec/ond?flag#hash | external

Browser without History API

Address: http://www.host.com/base/index.htm

$location.protocol() = http
$location.host() = www.host.com
$location.port() = 80
$location.path() = /path
$location.search() = {"a":"b"}
$location.hash() = h
/base/first?a=b | sec/ond?flag#hash | external

Caveats

Page reload navigation

The `$location` service allows you to change only the URL; it does not allow you to reload the page. When you need to change the URL and reload the page or navigate to a different page, please use a lower level API, `$window.location.href`.

Using $location outside of the scope life-cycle

`$location` knows about Angular's `scope` life-cycle. When a URL changes in the browser it updates the `$location` and calls `$apply` so that all $watchers / $observers are notified. When you change the `$location` inside the `$digest` phase everything is ok; `$location` will propagate this change into browser and will notify all the $watchers / $observers. When you want to change the `$location` from outside Angular (for example, through a DOM Event or during testing) - you must call `$apply` to propagate the changes.

$location.path() and ! or / prefixes

A path should always begin with forward slash (`/`); the `$location.path()` setter will add the forward slash if it is missing.

Note that the `!` prefix in the hashbang mode is not part of `$location.path()`; it is actually hashPrefix.

Testing with the $location service

When using `$location` service during testing, you are outside of the angular's `scope` life-cycle. This means it's your responsibility to call `scope.$apply()`.

```
describe('serviceUnderTest', function() {
  beforeEach(module(function($provide) {
    $provide.factory('serviceUnderTest', function($location){
      // whatever it does...
    });
  });

  it('should...', inject(function($location, $rootScope, serviceUnderTest) {
    $location.path('/new/path');
    $rootScope.$apply();

    // test whatever the service should do...

  }));
});
```

Migrating from earlier AngularJS releases

In earlier releases of Angular, `$location` used `hashPath` or `hashSearch` to process path and search methods. With this release, the `$location` service processes path and search methods and then uses the information it obtains to compose hashbang URLs (such as `http://server.com/#!/path?search=a`), when necessary.

Changes to your code

Navigation inside the app	Change to
$location.href = value $location.hash = value $location.update(value) $location.updateHash(value)	$location.path(path).search(search)
$location.hashPath = path	$location.path(path)
$location.hashSearch = search	$location.search(search)
Navigation outside the app	Use lower level API
$location.href = value $location.update(value)	$window.location.href = value
$location[protocol | host | port | path | search]	$window.location[protocol | host | port | path | search]
Read access	Change to
$location.hashPath	$location.path()
$location.hashSearch	$location.search()

Navigation inside the app	Change to
$location.href $location.protocol $location.host $location.port $location.hash	$location.absUrl() $location.protocol() $location.host() $location.port() $location.path() + $location.search()
$location.path $location.search	$window.location.path $window.location.search

Two-way binding to $location

The Angular's compiler currently does not support two-way binding for methods (see issue). If you should require two-way binding to the $location object (using `ngModel` directive on an input field), you will need to specify an extra model property (e.g. `locationPath`) with two watchers which push $location updates in both directions. For example:

```
<!-- html -->
<input type="text" ng-model="locationPath" />
```

```
// js - controller
$scope.$watch('locationPath', function(path) {
  $location.path(path);
});

$scope.$watch('$location.path()', function(path) {
  scope.locationPath = path;
});
```

Related API

- `$location API`

While Angular offers several useful services, for any nontrivial application you'll find it useful to write your own custom services. To do this you begin by registering a service factory function with a module either via the `Module#factory api` or directly via the `$provide` api inside of module config function.

All Angular services participate in dependency injection (DI) by registering themselves with Angular's DI system (injector) under a `name` (id) as well as by declaring dependencies which need to be provided for the factory function of the registered service. The ability to swap dependencies for mocks/stubs/dummies in tests allows for services to be highly testable.

Registering Services

To register a service, you must have a module that this service will be part of. Afterwards, you can register the service with the module either via the `Module api` or by using the `$provide` service in the module configuration function.The following pseudo-code shows both approaches:

Using the angular.Module api:

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

Using the $provide service:

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

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

Dependencies

Services can not only be depended upon, but can also have their own dependencies. These can be specified as arguments of the factory function. Read more about dependency injection (DI) in Angular and the use of array notation and the $inject property to make DI annotation minification-proof.

Following is an example of a very simple service. This service depends on the `$window` service (which is passed as a parameter to the factory function) and is just a function. The service simply stores all notifications; after the third one, the service displays all of the notifications by window alert.

```
angular.module('myModule', [], function($provide) {
  $provide.factory('notify', ['$window', function(win) {
    var msgs = [];
    return function(msg) {
      msgs.push(msg);
      if (msgs.length == 3) {
        win.alert(msgs.join("\n"));
        msgs = [];
      }
    };
  }]);
});
```

Instantiating Angular Services

All services in Angular are instantiated lazily. This means that a service will be created only when it is needed for instantiation of a service or an application component that depends on it. In other words, Angular won't instantiate services unless they are requested directly or indirectly by the application.

Services as singletons

Lastly, it is important to realize that all Angular services are application singletons. This means that there is only one instance of a given service per injector. Since Angular is lethally allergic to global state, it is possible to create multiple injectors, each with its own instance of a given service, but that is rarely needed, except in tests where this property is crucially important.

Related Topics

- Understanding Angular Services
- Managing Service Dependencies
- Injecting Services Into Controllers
- Testing Angular Services

Related API

- Angular Service API

Using services as dependencies for controllers is very similar to using services as dependencies for another service.

Since JavaScript is a dynamic language, DI can't figure out which services to inject by static types (like in static typed languages). Therefore, you can specify the service name by using the `$inject` property, which is an array containing strings with names of services to be injected. The name must match the corresponding service ID registered with angular. The order of the service IDs matters: the order of the services in the array will be used when calling the factory function with injected parameters. The names of parameters in factory function don't matter, but by convention they match the service IDs, which has added benefits discussed below.

```
function myController($loc, $log) {
this.firstMethod = function() {
 // use $location service
 $loc.setHash();
};
this.secondMethod = function() {
 // use $log service
 $log.info('...');
};
}
// which services to inject ?
myController.$inject = ['$location', '$log'];
```

Source

index.html script.js End to end test Edit

index.html :

```
<!doctype html>
<html ng-app="MyServiceModule">
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="myController">
      <p>Let's try this simple notify service, injected into the controller...</p>
      <input ng-init="message='test'" ng-model="message" >
      <button ng-click="callNotify(message);">NOTIFY</button>
    </div>
  </body>
</html>
```

script.js :

```
angular.
 module('MyServiceModule', []).
 factory('notify', ['$window', function(win) {
    var msgs = [];
    return function(msg) {
      msgs.push(msg);
      if (msgs.length == 3) {
        win.alert(msgs.join("\n"));
        msgs = [];
      }
    };
  }]);

function myController(scope, notifyService) {
  scope.callNotify = function(msg) {
    notifyService(msg);
  };
}

myController.$inject = ['$scope','notify'];
```

End to end test :

```
it('should test service', function() {
  expect(element(':input[ng\\:model="message"]').val()).toEqual('test');
});
```

Demo

Let's try this simple notify service, injected into the controller...

test NOTIFY

Implicit Dependency Injection

A new feature of Angular DI allows it to determine the dependency from the name of the parameter. Let's rewrite the above example to show the use of this implicit dependency injection of `$window`, `$scope`, and our `notify` service:

Source

index.html script.js Edit

index.html :

```
<!doctype html>
<html ng-app="MyServiceModuleDI">
  <head>
    <script src="http://code.angularjs.org/1.0.4angular.min.js"></script>
    <script src="script.js"></script>
  </head>
  <body>
    <div ng-controller="myController">
      <p>Let's try the notify service, that is implicitly injected into the controller...</p>
      <input ng-init="message='test'" ng-model="message">
      <button ng-click="callNotify(message);">NOTIFY</button>
    </div>
  </body>
</html>
```

script.js :

```
angular.
 module('MyServiceModuleDI', []).
 factory('notify', function($window) {
    var msgs = [];
    return function(msg) {
      msgs.push(msg);
      if (msgs.length == 3) {
        $window.alert(msgs.join("\n"));
        msgs = [];
      }
    };
  });

function myController($scope, notify) {
  $scope.callNotify = function(msg) {
    notify(msg);
  };
}
```

Demo

Let's try the notify service, that is implicitly injected into the controller...

However, if you plan to minify your code, your variable names will get renamed in which case you will still need to explicitly specify dependencies with the `$inject` property.

Related Topics

Understanding Angular Services Creating Angular Services Managing Service Dependencies Testing Angular Services

Related API

Angular Service API

Angular allows services to declare other services as dependencies needed for construction of their instances.

To declare dependencies, you specify them in the factory function signature and annotate the function with the inject annotations either using by setting the `$inject` property, as an array of string identifiers or using the array notation. Optionally the `$inject` property declaration can be dropped (see "Inferring `$inject`" but note that that is currently an experimental feature).

Using the array notation:

```
function myModuleCfgFn($provide) {
  $provide.factory('myService', ['dep1', 'dep2', function(dep1, dep2) {}]);
}
```

Using the $inject property:

```
function myModuleCfgFn($provide) {
  var myServiceFactory = function(dep1, dep2) {};
  myServiceFactory.$inject = ['dep1', 'dep2'];
  $provide.factory('myService', myServiceFactory);
}
```

Using DI inference (incompatible with minifiers):

```
function myModuleCfgFn($provide) {
  $provide.factory('myService', function(dep1, dep2) {});
}
```

Here is an example of two services, one of which depends on the other and both of which depend on other services that are provided by the Angular framework:

```
/**
 * batchLog service allows for messages to be queued in memory and flushed
 * to the console.log every 50 seconds.
 *
 * @param {*} message Message to be logged.
 */
  function batchLogModule($provide){
    $provide.factory('batchLog', ['$timeout', '$log', function($timeout, $log) {
      var messageQueue = [];

      function log() {
        if (messageQueue.length) {
          $log('batchLog messages: ', messageQueue);
          messageQueue = [];
        }
        $timeout(log, 50000);
      }

      // start periodic checking
      log();

      return function(message) {
        messageQueue.push(message);
      }
    }]);

    /**
     * routeTemplateMonitor monitors each $route change and logs the current
     * template via the batchLog service.
     */
    $provide.factory('routeTemplateMonitor',
                ['$route', 'batchLog', '$rootScope',
         function($route,   batchLog,   $rootScope) {
      $rootScope.$on('$routeChangeSuccess', function() {
        batchLog($route.current ? $route.current.template : null);
      });
    }]);
  }

  // get the main service to kick of the application
  angular.injector([batchLogModule]).get('routeTemplateMonitor');
```

Things to notice in this example:

- The `batchLog` service depends on the built-in `$timeout` and `$log` services, and allows messages to be logged into the `console.log` in batches.
- The `routeTemplateMonitor` service depends on the built-in `$route` service as well as our custom `batchLog` service.
- Both of our services use the factory function signature and array notation for inject annotations to declare their dependencies. It is important that the order of the string identifiers in the array is the same as the order of argument names in the signature of the factory function. Unless the dependencies are inferred from the function signature, it is this array with IDs and their order that the injector uses to determine which services and in which order to inject.

Related Topics

- Understanding Angular Services
- Creating Angular Services
- Injecting Services Into Controllers
- Testing Angular Services

Related API

- Angular Service API
- `Angular Injector API`

The following is a unit test for the 'notify' service in the 'Dependencies' example in Creating Angular Services. The unit test example uses Jasmine spy (mock) instead of a real browser alert.

```
var mock, notify;

beforeEach(function() {
  mock = {alert: jasmine.createSpy()};

  module(function($provide) {
    $provide.value('$window', mock);
  });

  inject(function($injector) {
    notify = $injector.get('notify');
  });
});

it('should not alert first two notifications', function() {
  notify('one');
  notify('two');

  expect(mock.alert).not.toHaveBeenCalled();
});

it('should alert all after third notification', function() {
  notify('one');
  notify('two');
  notify('three');

  expect(mock.alert).toHaveBeenCalledWith("one\ntwo\nthree");
});

it('should clear messages after alert', function() {
  notify('one');
  notify('two');
  notify('third');
  notify('more');
  notify('two');
  notify('third');

  expect(mock.alert.callCount).toEqual(2);
  expect(mock.alert.mostRecentCall.args).toEqual(["more\ntwo\nthird"]);
});
```

Related Topics

- Understanding Angular Services
- Creating Angular Services
- Managing Service Dependencies
- Injecting Services Into Controllers

Related API

- Angular Service API

Angular services are singletons that carry out specific tasks common to web apps, such as the `$http service` that provides low level access to the browser's `XMLHttpRequest` object.

To use an Angular service, you identify it as a dependency for the dependent (a controller, or another service) that depends on the service. Angular's dependency injection subsystem takes care of the rest. The Angular injector subsystem is in charge of service instantiation, resolution of dependencies, and provision of dependencies to factory functions as requested.

Angular injects dependencies using "constructor" injection (the service is passed in via a factory function). Because JavaScript is a dynamically typed language, Angular's dependency injection subsystem cannot use static types to identify service dependencies. For this reason a dependent must explicitly define its dependencies by using the `$inject` property. For example:

```
    myController.$inject = ['$location'];
```

The Angular web framework provides a set of services for common operations. Like other core Angular variables and identifiers, the built-in services always start with `$` (such as `$http` mentioned above). You can also create your own custom services.

Related Topics

- About Angular Dependency Injection
- Creating Angular Services
- Managing Service Dependencies
- Testing Angular Services

Related API

- Angular Service API
- `Injector API`

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

It is all about NOT mixing concerns

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

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

While this may seem obvious it usually is very difficult to be able to call an individual function on a typical project. The reason is that the developers often mix concerns, and they end up with a piece of code which does everything. It reads the data from XHR, it sorts it and then it manipulates the DOM.

With Angular we try to make it easy for you to do the right thing, and so we provide dependency injection for your XHR (which you can mock out) and we created abstraction which allow you to sort your model without having to resort to manipulating the DOM. So that in the end, it is easy to write a sort function which sorts some data, so that your test can create a data set, apply the function, and assert that the resulting model is in the correct order. The test does not have to wait for XHR, or create the right kind of DOM, or assert that your function has mutated the DOM in the right way.

With great power comes great responsibility

Angular is written with testability in mind, but it still requires that you do the right thing. We tried to make the right thing easy, but Angular is not magic, which means if you don't follow these guidelines you may very well end up with an untestable application.

Dependency Injection

There are several ways in which you can get a hold of a dependency: 1. You could create it using the `new` operator. 2. You could look for it in a well known place, also known as global singleton. 3. You could ask a registry (also known as service registry) for it. (But how do you get a hold of the registry? Most likely by looking it up in a well known place. See #2) 4. You could expect that it be handed to you.

Out of the four options in the list above, only the last one is testable. Let's look at why:

Using the `new` operator

While there is nothing wrong with the `new` operator fundamentally the issue is that calling a new on a constructor permanently binds the call site to the type. For example lets say that we are trying to instantiate an `XHR` so that we can get some data from the server.

```
function MyClass() {
  this.doWork = function() {
    var xhr = new XHR();
    xhr.open(method, url, true);
    xhr.onreadystatechange = function() {...}
    xhr.send();
  }
}
```

The issue becomes that in tests, we would very much like to instantiate a `MockXHR` which would allow us to return fake data and simulate network failures. By calling `new XHR()` we are permanently bound to the actual XHR, and there is no good way to replace it. Yes there is monkey patching. That is a bad idea for many reasons which are outside the scope of this document.

The class above is hard to test since we have to resort to monkey patching:

```
var oldXHR = XHR;
XHR = function MockXHR() {};
var myClass = new MyClass();
myClass.doWork();
// assert that MockXHR got called with the right arguments
XHR = oldXHR; // if you forget this bad things will happen
```

Global look-up:
Another way to approach the problem is to look for the service in a well known location.

```
function MyClass() {
  this.doWork = function() {
    global.xhr({
      method:'...',
      url:'...',
      complete:function(response){ ... }
    })
  }
}
```

While no new instance of the dependency is being created, it is fundamentally the same as `new`, in that there is no good way to intercept the call to `global.xhr` for testing purposes, other then through monkey patching. The basic issue for testing is that global variable needs to be mutated in order to replace it with call to a mock method. For further explanation why this is bad see: Brittle Global State & Singletons

The class above is hard to test since we have to change global state:

```
var oldXHR = global.xhr;
global.xhr = function mockXHR() {};
var myClass = new MyClass();
myClass.doWork();
// assert that mockXHR got called with the right arguments
global.xhr = oldXHR; // if you forget this bad things will happen
```

Service Registry:
It may seem as that this can be solved by having a registry for all of the services, and then having the tests replace the services as needed.

```
function MyClass() {
  var serviceRegistry = ????;
  this.doWork = function() {
    var xhr = serviceRegistry.get('xhr');
    xhr({
      method:'...',
      url:'...',
      complete:function(response){ ... }
    })
}
```

However, where does the serviceRegistry come from? if it is: * new-ed up, the the test has no chance to reset the services for testing * global look-up, then the service returned is global as well (but resetting is easier, since there is only one global variable to be reset).

The class above is hard to test since we have to change global state:

```
var oldServiceLocator = global.serviceLocator;
global.serviceLocator.set('xhr', function mockXHR() {});
var myClass = new MyClass();
myClass.doWork();
// assert that mockXHR got called with the right arguments
global.serviceLocator = oldServiceLocator; // if you forget this bad things will happen
```

Passing in Dependencies:
Lastly the dependency can be passed in.

```
function MyClass(xhr) {
  this.doWork = function() {
    xhr({
      method:'...',
      url:'...',
      complete:function(response){ ... }
    })
}
```

This is the preferred way since the code makes no assumptions as to where the `xhr` comes from, rather that whoever created the class was responsible for passing it in. Since the creator of the class should be different code than the user of the class, it separates the responsibility of creation from the logic, and that is what dependency-injection is in a nutshell.

The class above is very testable, since in the test we can write:

```
function xhrMock(args) {...}
var myClass = new MyClass(xhrMock);
myClass.doWork();
// assert that xhrMock got called with the right arguments
```

Notice that no global variables were harmed in the writing of this test.

Angular comes with dependency injection built in which makes the right thing easy to do, but you still need to do it if you wish to take advantage of the testability story.

Controllers

What makes each application unique is its logic, which is what we would like to test. If the logic for your application is mixed in with DOM manipulation, it will be hard to test as in the example below:

```
function PasswordCtrl() {
  // get references to DOM elements
  var msg = $('.ex1 span');
  var input = $('.ex1 input');
  var strength;

  this.grade = function() {
    msg.removeClass(strength);
    var pwd = input.val();
    password.text(pwd);
    if (pwd.length > 8) {
      strength = 'strong';
    } else if (pwd.length > 3) {
      strength = 'medium';
    } else {
      strength = 'weak';
    }
    msg
     .addClass(strength)
     .text(strength);
  }
}
```

The code above is problematic from a testability point of view, since it requires your test to have the right kind of DOM present when the code executes. The test would look like this:

```
var input = $('<input type="text"/>');
var span = $('<span>');
$('body').html('<div class="ex1">')
  .find('div')
    .append(input)
    .append(span);
var pc = new PasswordCtrl();
input.val('abc');
pc.grade();
expect(span.text()).toEqual('weak');
$('body').html('');
```

In angular the controllers are strictly separated from the DOM manipulation logic which results in a much easier testability story as can be seen in this example:

```
function PasswordCtrl($scope) {
  $scope.password = '';
  $scope.grade = function() {
    var size = $scope.password.length;
    if (size > 8) {
      $scope.strength = 'strong';
    } else if (size > 3) {
      $scope.strength = 'medium';
    } else {
      $scope.strength = 'weak';
    }
  };
}
```

and the test is straight forward

```
var pc = new PasswordCtrl();
pc.password('abc');
pc.grade();
expect(pc.strength).toEqual('weak');
```

Notice that the test is not only much shorter but it is easier to follow what is going on. We say that such a test tells a story, rather then asserting random bits which don't seem to be related.

Filters

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

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

var length = $filter('length');
expect(length(null)).toEqual(0);
expect(length('abc')).toEqual(3);
```

Directives

Directives in angular are responsible for updating the DOM when the state of the model changes.

Mocks

oue

Global State Isolation

oue

Preferred way of Testing

uo

JavaScriptTestDriver

ou

Jasmine

ou

Sample project

uoe