Overview

Angular's [HTML compiler](https://docs.angularjs.org/api/ng/service/$compile) allows the developer to teach the browser new HTML syntax. The compiler allows you to attach behavior to any HTML element or attribute and even create new HTML elements or attributes with custom behavior. Angular calls these behavior extensions [directives](https://docs.angularjs.org/api/ng/provider/$compileProvider#directive).

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

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

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 extensions become a Domain Specific Language for building your application.

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

Compiler

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

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

Some directives such as [ng-repeat](https://docs.angularjs.org/api/ng/directive/ngRepeat) clone DOM elements once for each item in a collection. Having a compile and link phase improves performance since the cloned template only needs to be compiled once, and then linked once for each clone instance.

Directive

A directive is a behavior which should be triggered when specific HTML constructs are encountered during the compilation process. The directives can be placed in element names, attributes, class names, as well as comments. Here are some equivalent examples of invoking the [ng-bind](https://docs.angularjs.org/api/ng/directive/ngBind) 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](https://docs.angularjs.org/api/ng/provider/$compileProvider#directive) 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.

  Edit in Plunker

[**script.jsindex.html**](https://docs.angularjs.org/)

angular.module('drag', []).

directive('draggable', function($document) {

return function(scope, element, attr) {

var startX = 0, startY = 0, x = 0, y = 0;

element.css({

position: 'relative',

border: '1px solid red',

backgroundColor: 'lightgrey',

cursor: 'pointer',

display: 'block',

width: '65px'

});

element.on('mousedown', function(event) {

// Prevent default dragging of selected content

event.preventDefault();

startX = event.screenX - x;

startY = event.screenY - y;

$document.on('mousemove', mousemove);

$document.on('mouseup', mouseup);

});

function mousemove(event) {

y = event.screenY - startY;

x = event.screenX - startX;

element.css({

top: y + 'px',

left: x + 'px'

});

}

function mouseup() {

$document.off('mousemove', mousemove);

$document.off('mouseup', mouseup);

}

};

});

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

Understanding View

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



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

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

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



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

How directives are compiled

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

HTML compilation happens in three phases:

1. [$compile](https://docs.angularjs.org/api/ng/service/$compile) traverses the DOM and matches directives.

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

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

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

1. $compile links the template with the scope by calling the combined linking function from the previous step. This in turn will call the linking function of the individual directives, registering listeners on the elements and setting up [$watchs](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$watch) with the [scope](https://docs.angularjs.org/api/ng/type/$rootScope.Scope) as each directive is configured to do.

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

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

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

var scope = ...;

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

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

// Step 1: parse HTML into DOM element

var template = angular.element(html);

// Step 2: compile the template

var linkFn = $compile(template);

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

var element = linkFn(scope);

// Step 4: Append to DOM (optional)

parent.appendChild(element);

The difference between Compile and Link

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

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

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

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

An Example of "Compile" Versus "Link"

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

Hello {{user.name}}, you have these actions:

<ul>

<li ng-repeat="action in user.actions">

{{action.description}}

</li>

</ul>

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

{{user.name}} matches the [interpolation directive](https://docs.angularjs.org/api/ng/service/$interpolate) and ng-repeat matches the [ngRepeat directive](https://docs.angularjs.org/api/ng/directive/ngRepeat).

But [ngRepeat](https://docs.angularjs.org/api/ng/directive/ngRepeat) has a dilemma.

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

As new actions are inserted, the template <li> element needs to be cloned and inserted into ul. But cloning the <li> element is not enough. It also needs to compile the <li> so that its directives, like {{action.description}}, evaluate against the right [scope](https://docs.angularjs.org/api/ng/type/$rootScope.Scope).

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

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

the **compile phase** where all of the directives are identified and sorted by priority, and a **linking phase** where any work which "links" a specific instance of the [scope](https://docs.angularjs.org/api/ng/type/$rootScope.Scope) and the specific instance of an <li> is performed.

**Note:** *Link* means setting up listeners on the DOM and setting up $watch on the Scope to keep the two in sync.

[ngRepeat](https://docs.angularjs.org/api/ng/directive/ngRepeat) works by preventing the compilation process from descending into the <li> element so it can make a clone of the original and handle inserting and removing DOM nodes itself.

Instead the [ngRepeat](https://docs.angularjs.org/api/ng/directive/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](https://docs.angularjs.org/api/ng/directive/ngRepeat) watches the expression and as items are added to the array it clones the <li> element, creates a new [scope](https://docs.angularjs.org/api/ng/type/$rootScope.Scope)for the cloned <li> element and calls the link function on the cloned <li>.

Understanding How Scopes Work with Transcluded Directives

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

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

<div>

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

<dialog title="Hello {{username}}."

visible="show"

on-cancel="show = false"

on-ok="show = false; doSomething()">

Body goes here: {{username}} is {{title}}.

</dialog>

</div>

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

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

<div ng-show="visible">

<h3>{{title}}</h3>

<div class="body" ng-transclude></div>

<div class="footer">

<button ng-click="onOk()">Save changes</button>

<button ng-click="onCancel()">Close</button>

</div>

</div>

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

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

scope: {

title: '@', // the title uses the data-binding from the parent scope

onOk: '&', // create a delegate onOk function

onCancel: '&', // create a delegate onCancel function

visible: '=' // set up visible to accept data-binding

}

Creating local properties on widget scope creates two problems:

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

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

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

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

Therefore the final directive definition looks something like this:

transclude: true,

scope: {

title: '@', // the title uses the data-binding from the parent scope

onOk: '&', // create a delegate onOk function

onCancel: '&', // create a delegate onCancel function

visible: '=' // set up visible to accept data-binding

},

restrict: 'E',

replace: true