MENU

ANGULAR TEMPLATE SYNTAX DEMYSTIFIED - PART 1

*by **Pascal Precht** on Aug 11, 2015, last updated on Dec 16, 2016*
13 minute read

I think we've been all through this. We see Angular template code the very first time and all of a sudden we have these weird brackets and parentheses spread all over our HTML. Around a year ago I've written an article about how to integrate Web Components with AngularJS, which explains how we can use, or not use, Web Components in AngularJS applications today. If you haven't read it yet, I highly recommend you doing so. It's old but the content is still true.

It also touches on the new template syntax in Angular and how it tends to solve the existing issues. However, it still seems a mystery for a lot of people and that's why we're going to explore the Angular's template syntax in this article. **Please note** that this is the first part of "Angular Template Syntax Demystified". There's going to be another article soon.

TABLE OF CONTENTS

- What it looks like
- Property Binding
- Event Binding
- Two-way Data Binding

What it looks like

Just to make sure that everyone knows what we are talking about when mentioning “weird” brackets and parentheses in our HTML, we’re going to make a quick recap on what this template syntax looks like.

If you’ve read the step-by-step guide you’ve probably seen the following syntaxes:

```
<p>My name: {{ myName}}</p>
<ul>
  <li *ngFor="let name of names">
    ...
  </li>
</ul>
<input #myname (keyup)="myControllerMethod()">
```

And that’s not all, in fact we could add some more:

```
<todo [todo]="todo"></todo>
<input [(ngModel)]="foo">
```

What we see here is **Event Binding**, **Property Binding**, **Local Variables** and **Template Directives**. In this article we’re going to focus on property, event and two-way binding. But before we explore all of these different syntaxes, let’s answer the first question that comes to our mind when seeing this code:

Is that valid HTML?

The answer is **yes.** Here’s an excerpt of the HTML Syntax Spec:

> *Attribute names must consist of one or more characters other than the space characters, U+0000 NULL, """, "'", ">", "/", "=", the control characters, and any characters that are not defined by Unicode.*

Which basically means, any other characters than the ones listed can be used in HTML attribute names. Let's start off by taking a look at the most important syntax for property binding.

Property Binding

Property binding is the syntax where we use brackets to bind values to an element's property. But what does that mean? Why do we want to bind to an element's property? Well, in order to understand why we want to do that, we have to go back to the basics and understand the APIs of a DOM element.

The APIs of a DOM element are:

- **Attributes** - Attributes are the things we use in HTML to provide elements with data. In fact, attributes are the only way in plain HTML to put values into an element. However, the type of an attribute value in HTML is always `String`, which is not always what we want especially when building applications with frameworks like Angular.
- **Properties** - Properties are simply the properties of a DOM object. E.g. if we query a DOM element with `document.querySelector()`, we get a DOM object back which simply has its own properties and methods. Those properties are no special in any way, they behave like any other object properties in JavaScript. That also means we can assign any kind of value to a property, not just strings.
- **Methods** - As already mentioned, methods are just the functions on a DOM object that we can call and execute in JavaScript. `setAttribute()` for example is such a method.
- **Events** - Of course, last but not least, we have events. The bread and butter when it comes to notifying subscribers that something happened. Like `click`, `focus` or `input`. DOM elements can also fire their own custom events.

The most interesting part here, is how attributes and properties behave on a DOM object. Especially when we realize, that they behave differently across elements.

Let's just take this simple `input` example here:

```
<input value="thoughtram">
```

We use the `value` attribute to set the **initial** value of the input element. Let's see what happens when we access the DOM object and it's `value` property:

```
var input = document.querySelector('input');
input.value // 'thoughtram'
```

Alright, we access the `value` property and as expected, it returns the string `thoughtram`. But what happens when we change that value and read from its attribute?

```
input.value = 'Angular Master Class';
input.getAttribute('value'); // 'thoughtram'
```

As we can see, the property value is not reflected back to the attribute. There are very few elements that actually reflect their property value back to its attribute. For example the `src` property of an `img` element. Changing its property will also change its attribute. In addition to that, a property can really get any value, whereas an attribute is always a string.

So how could we pass objects to directives in AngularJS? Well, as most of us know, there's this directive definition object (DDO), which allows us to specify how directives' scope properties (or controller properties) are bound to the outside world. The following code for example, allows us to pass an expression to a directive via attributes, that actually results in an object once evaluated.

```
angular.module('app', []).directive('widget', function () {
  return {
    scope: {},
    bindToController: {
      obj: '='
    },
    controller: function () {},
    controllerAs: 'ctrl',
    template: 'Value: ctrl.obj.foo'
  }
});
```

If `bindToController` is new to you, you might want to read this article.

From the outside world, we can then pass an object to our `widget` directive simply with an expression. Let's say we have a controller like this:

```
angular.module('app', []).controller('AppController', function () {
  this.objOnCtrl = {
    foo: 'Hello World!'
  };
});
```

And a template like this:

```
<widget ng-controller="AppController as ctrl" obj="ctrl.objOnCtrl">
```

Angular takes the expression passed to the attribute and parses and evaluates it against the corresponding scope, which allows us to pass other values than strings to directives.

Why can't we just continue like that in Angular? There are a couple of reasons:

- **Predictability** - Looking at our template, we can't tell what happens to the value of the attribute inside our directive, without knowing the internals of it. We need a way to decide from the outside world how values are bound.
- **Compatibility** - This whole mechanism only works with Angular directives. As soon as we use custom elements that aren't directives but vanilla Web Components, `ctrl.objOnCtrl` would just be that string, unless our custom element would come with a similar parsing and evaluating strategy that Angular directives use. Angular should work with any element, no matter if it's a custom element, a Web Component built with Polymer, or a directive.

That's why since version 2.x, Angular always binds to properties rather than attributes (as AngularJS does). Every DOM element has properties, regardless of being a native element or a Web Component. In order to tell Angular that we want to bind to a property, we use the brackets syntax. If we'd build our `widget` directive in Angular, we would need to bind to it's `obj` property to assign an object value:

```
<widget [obj]="objOnComponent">
```

Another feature that property binding brings to the table is **escaping**. Just think about the `img` tag and its `src` attribute. What happens when the browser ecounters an `img` tag with a source when parsing the HTML? Right, it tries to request the source of that image. If we'd have HTML like this:

```
<img src="{{someExpression}}">
```

It would result in a `404` error, because the source isn't a valid URL. That's why AngularJS has this `ng-src` directive. It takes an expression, evaluates it and once evaluated, it adds the actual `src` attribute to the image element, which results in a correct request. Binding to properties escapes automatically, because value will also only be assigned once their expressions are evaluated.

We can still use normal attributes without brackets though. Just keep in mind that normal attributes behave like normal attributes.

What about interpolations?

Good question! Interpolation in Angular has always been one of the biggest features of the framework. If there's only property binding, how does something like the following work then?

```
<p>Hello {{name}}</p>
```

It turns out, that interpolation syntax is just a shorthand syntax for property binding as well. The code above could be translated to something like this:

```
<p [textContent]="interpolate(['Hello'], [name])"></p>
```

Canonical syntax

Okay cool, now we know what property binding is and why we want to use it, but there's still a problem we haven't talked about yet. What if we generate our templates with an HTML preprocessor that doesn't like the brackets syntax? Guess what, the brackets syntax itself is really just a shorthand for us developers so we can save some key strokes. All property bindings inside a template can be written as:

```
<ANY bind-{PROPERTY_NAME}="{EXPRESSION}"></ANY>
```

Which makes it an alpha numeric version that all preprocessors should be able to deal with.

Event Binding

Binding to properties through HTML is already super powerful. But sometimes we need to be able to react to certain things that happen during runtime of our application. This is what DOM events are for and they are part of our browser platforms since years. Just think of `click`, `input` and `focus` events. However, nowadays it's also very common for elements to fire their own **custom events**. A Web Component `<date-picker>` for example, could fire a `dateChanged` event.

In AngularJS, we have a lot of directives that help us out notifying the framework when an event is fired. That's why we have things like `ng-click`, `ng-focus` etc. They simply notify Angular that an event has been fired and application state could have changed. Without these directives, Angular's two-way data binding wouldn't work out-of-the-box. Unfortunately, that doesn't really scale. How can our `<date-picker>` Web Component notify Angular that a change has happened? Correct, we would need to create a directive for each and every event and it's not uncommon for elements to fire more than just one event.

That's why since version 2.x, in Angular we have the parenthesis syntax where we can bind to **any** event like this:

```
<date-picker (dateChanged)="statement()"></date-picker>
```

The parentheses simply tell Angular that the expression inside the symbols is an event name that it needs to add an event listener for. `statement()` is simply the statement expression that gets executed whenever such an event is fired on that element. Having such a generic binding, all directives that intercepted for us in AngularJS will go away in Angular. This framework is getting simpler and simpler.

Event Bubbling

When binding to events in Angular, events are caught on the same element as well as from events that bubble up from child elements. E.g. if we have a DOM structure like this:

```
<div (click)="statement()">
  <div></div>
</div>
```

This will execute the given statement, if the first `div` or any of it's child elements are clicked.

Canonical syntax

Of course, using parenthesis is also just a shorthand syntax, so we as developers can save some key strokes. We can always use the canonical syntax, which is `on-*`.

```
<ANY on-{EVENT_NAME}="{STATEMENT}"></ANY>
```

Two-way Data Binding

We talked about property binding and event binding but what about the biggest selling point of AngularJS - two-way data binding?

Since 2.x, Angular doesn't come with two-way data binding by default. This feature has been removed intentionally from the framework, because it comes with various drawbacks. But how is something like `ng-model` in Angular implemented then? `ng-model` in Angular version >= 2.x gives us two-way data binding, even though under the hood, it simply uses a combination of property and event binding.

Let's take a look at how that works with the following snippet:

```
<input [value]="name">

<p>Hello {{name}}</p>
```

We already learned what's happening here. We're simply binding the value of a `name` expression to the `value` property of an input element. In addition we want to output the same value in our view and update it when someone changes the value by typing into the input. This, however, doesn't work because we're just writing to the elements property. There's nothing that tells the expression to update when the value changes through typing.

This can be easily fixed by adding an event binding accordingly:

```
<input [value]="name" (input)="name = $event.target.value">

<p>Hello {{name}}</p>
```

We listen to the `input` event and whenever it is fired, we update the value of our `name` expression by pulling the new value out of the event object.

Since this is quite a lot of typing to implement such a common scenario, Angular comes with an `ngModel` directive that saves some key strokes for us, by unifying the property and event binding, and also the extraction of the target's value. Here's the exact same code using `ngModel`:

```
<input [ngModel]="name" (ngModelChange)="name = $event">

<p>Hello {{name}}</p>
```

It gets even better! We can shorten that syntax even further by merging both bindings like this:

```
<input [(ngModel)]="name">

<p>Hello {{name}}</p>
```

Two-way data binding is back in town! We can easily build our own directives that support that syntax, but that we're going to explore in another article.

This was the first part of "Angular Template Syntax Demystified". We're going to cover local variables and template directives in a second part, which will be published very soon.

SUBSCRIBE

Information on the performance measurement included in the consent, the use of the mail service provider MailChimp and on the logging of the registration and your rights of revocation can be found in our data protection declaration.

Share on Facebook Share on Twitter Share on Google+

AUTHOR

Pascal Precht

Pascal is a front-end engineer and a Angular Developer Expert nominated by Google. He created the angular-translate module, is an Angular contributor and also part of the Angular Docs Authoring team.

Twitter GitHub

RELATED POSTS

Advanced caching with RxJS

When building web applications, performance should always be a top priority. One very efficient way to optimize the performance of...

Custom Overlays with Angular's CDK - Part 2

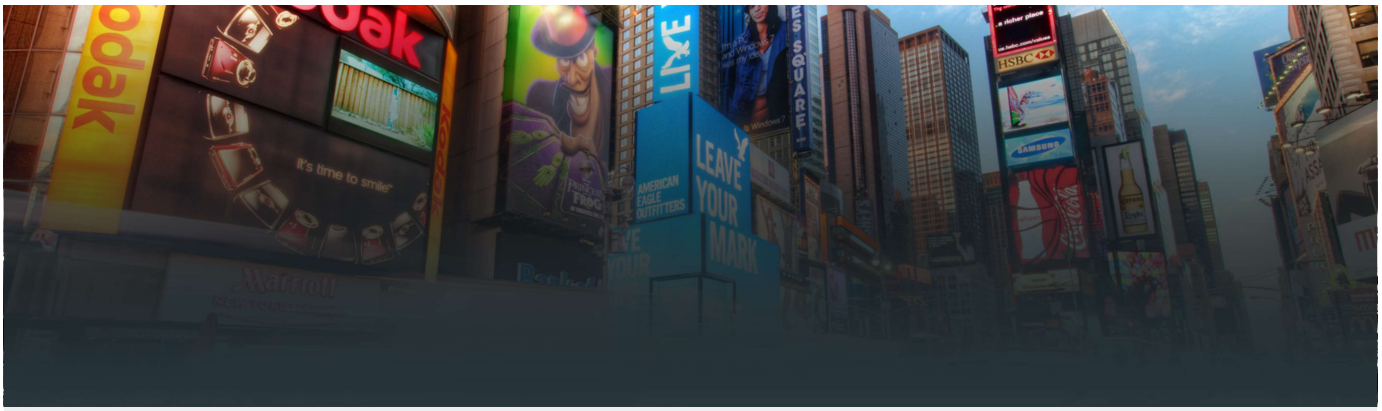
In this follow-up post we demonstrate how to use Angular's CDK to build a custom overlay that looks and feels...

Custom Overlays with Angular's CDK

The Angular Material CDK provides us with tools to build awesome and high-quality Angular components without adopting the Material Design...

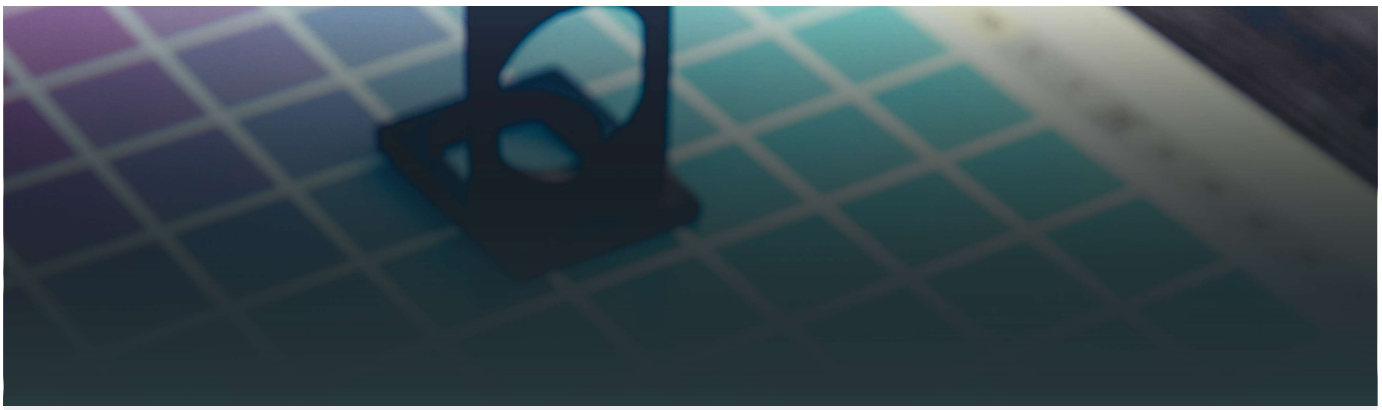
Easy Dialogs with Angular Material

Building modals and dialogs isn't easy - if we do it ourselves. Angular Material comes with a powerful dialog service...



A web animations deep dive with Angular

Angular comes with a built-in animation system that lets us create powerful animations based on the Web Animations API. In...



Custom themes with Angular Material

Angular Material offers great theming capabilities for both, built-in and custom themes. In this article we'll explore how to make...