one way binding vs two way binding

Every HTML attribute specific to AngularJS is called **directive** and has a specific role in an AngularJS application.

We say that the HTML page is a **template** because it's not the final version of what will be rendered by the web browser and visible to the user, but it's in fact just the structure of the web page before it's processed by AngularJS.

The ng-app directive marks the DOM element that contains the AngularJS application. We initialize a couple of variables firstName and lastName through the ng-init directive.

A data binding can be specified in two different ways:

- with curly braces: {{*expression*}}
- with the ng-bind directive: ng-bind="varName"

We're saying **one way data binding** because the model values (here the variables represent the model) are automatically assigned to the HTML placeholder elements specified through the data binding notation, but the HTML elements don't change the values in the model (one way).

```
<!DOCTYPE html>
<html>
<head>
  <script src="angular.js"></script>
</head>

<body ng-app ng-init="firstName = 'John'; lastName = 'Doe';">
  <strong>First name:</strong> {{firstName}}<br />
  <strong>Last name:</strong> <span ng-bind="lastName"></span>
</body>
</html>
```

First name: John
Last name: Doe

We have a **two way data binding** when a model variable is bound to a HTML element that can both change and display the value of the variable. In general, we could have more than one HTML element bound to the same variable.
We use the ng-model directive to bind a model variable to the HTML element that can not only display its value, but also change it.
In the example, we bind the firstName and lastName model variables to a couple of HTML input elements. When the page is loaded, the value of the input elements are initialized to those of the respective model variables and whenever the user types something in an input, the value of the model variable is modified as well (two way).

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

This interpolation directive registers a watch for a property `name` on the corresponding scope (which in our case is `$rootScope`) in order to interpolate against it to display the value in the DOM.
Defining a property with exactly that identifier on our scope and assigning a value to it, makes it magically displaying it in the DOM without further actions:

```
angular.module('myApp', [])
.run(function ($rootScope) {
  $rootScope.name = "Pascal";
});
```

Great! We just bound a model value to the view with an interpolation directive. If now something changes the value, the view gets updated automatically. Let's add a button that updates the value of `name` once it's clicked:

```
<button ng-click="name = 'Christoph'">Click me!</button>
```

Clicking the button assigns the string `Christoph` to `name` which triggers a$digest cycle that automatically updates the DOM accordingly. In this particular case we're just updating the value one-way (top → down).
However, when for example dealing with an `input` element that has ngModel directive applied, and a user changes its `value` property by typing something into it, the change is reflected back to the actual model.
This happens because when a `$digest` cycle is triggered, Angular processes all registered watchers on the current scope and its children and checks for

model mutations and calls dedicated watch listeners until the model is stabilized and no more listeners are fired. Once the `$digest` loop finishes the execution, the browser re-renders the DOM and reflects the changes.

The problem with too many watchers

Now that we have a picture of how the databinding mechanism in Angular actually works, we might wonder why there is a feature for one-time binding. Due to Angulars nature of using watchers for databinding, we might get some problems in terms of performance when having too many of them. As we learned, *watch expressions* are registered on the scope together with their callback listeners so Angular can process them during `$digest` cycles in order to update the view accordingly. That simply means, the more watchers are registered, the more Angular has to process.

Now imagine you have a lot of dynamic values in your view that have to be evaluated by Angular. Internationalization for example, is a very common use case where developers use Angulars databinding to localize their apps, even if the language isn't changeable during runtime, but set on initial page load. In that case every single string that is localized in the view and written to the scope, sets up a watch in order to get updated once something triggers the next `$digest`. This is a lot of overhead especially when your language actually doesn't change at runtime.

https://docs.angularjs.org/guide/expression#one-time-binding

Using this new syntax is as easy as starting an expression with `::`. So if we apply the one-time expression to our example above, we change this:

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

To this:

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

This works for all kind of typical Angular expressions you're used to use throughout your app. Which means you can use them in `ng-repeat`expressions or even for directives that expose attributes that set up a two-way binding from the inside out.